1.0 INTRODUCTION: SOFTWARE ISSUES

The term "*reliable*" is part of everyday vocabulary. Its meaning is somewhat intuitive. People talk about people, machines, processes and even software as being reliable. Typically "reliable" is equivalent to "trustworthy", or "dependable" or "true". In general, something is reliable if it meets the user's need under specified conditions. The elements of consistency (i.e., repeatedly obtaining the same result) and stability (meeting the need over time) are also important aspects of the intuitive notion of reliability.

1.1 A LITTLE RELIABILITY HISTORY

Hardware reliability engineering was first introduced as a discipline during World War II to evaluate the probability of success of ballistic rockets. The 1950s brought more advanced methods to estimate life expectancies of mechanical, electrical and electronic components used in the defense and aerospace industry. By the 1960s, reliability engineering had established itself as an integral part of end user product development in commercial products as well as military applications.[1]

The *software reliability* discipline is much younger, beginning in the mid 1970s when the software development environment was reasonably stable. Most of software reliability models discussed in this review were developed during this time of software stability. However, a surge of new technology, new paradigms, new structured analysis concepts, and new ways of developing software emerged in the late 1980s and continues to this date. Figure 1-1 provides a chronological reference for some of the elements which comprise the current software development environment and add to its complexity. Some recent efforts[2] have incorporated this changing software development environment into the reliability prediction model.

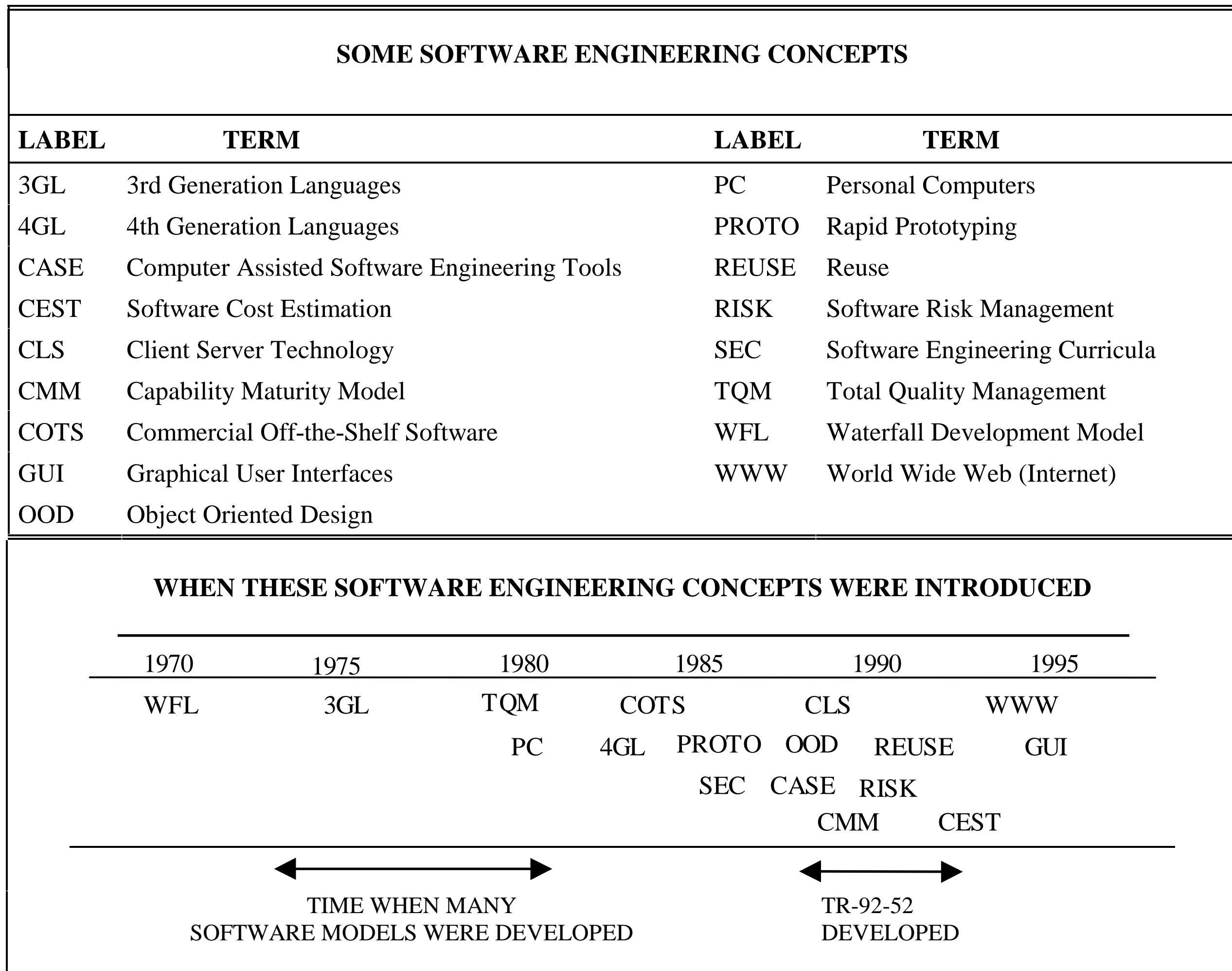
1.2 WHY CONSIDER SOFTWARE RELIABILITY?

Fielded systems, containing both hardware and software, have become very complex while significant cost constraints have been imposed and time-to-field requirements decreased. As software becomes a larger portion of the total cost, requirements for error-free, fault tolerant software have been instituted. For example, as both military and commercial airplanes move to fly-by-wire technology without mechanical back-up subsystems and without the ability to manually override the software with hardware subsystems, more assurance must be specified for the imbedded controlling software code. This, in turn, requires the achievement of very high software reliability, i.e., an extremely high confidence in the ability of the software to perform flawlessly. This has led to the development of software reliability metrics and models to quantify software reliability.

---

[1] See Coppola (1984) for a detailed history of hardware reliability.

[2] For example, the model published in 1992 by Rome Laboratory (TR-92-52).



SOME SOFTWARE ENGINEERING CONCEPTS

LABEL	TERM	LABEL	TERM
3GL	3rd Generation Languages	PC	Personal Computers
4GL	4th Generation Languages	PROTO	Rapid Prototyping
CASE	Computer Assisted Software Engineering Tools	REUSE	Reuse
CEST	Software Cost Estimation	RISK	Software Risk Management
CLS	Client Server Technology	SEC	Software Engineering Curricula
CMM	Capability Maturity Model	TQM	Total Quality Management
COTS	Commercial Off-the-Shelf Software	WFL	Waterfall Development Model
GUI	Graphical User Interfaces	WWW	World Wide Web (Internet)
OOD	Object Oriented Design		

Figure 1-1. Software Environment Timeline

As more and more systems that are a part of everyday life become more and more dependent upon software, perceptions about software reliability have changed. Increasing control by software of items such as dishwashers, ovens and automobiles, along with liability issues associated with these products, has led to an increased awareness of the criticality of reducing "hidden" software errors. Additionally, the influx of computers into financial and security-related operations requires a guarantee of data integrity.

1.3 RELIABILITY DEFINITIONS

Hardware reliability is defined as the probability that an item will perform its intended function for a specified interval under stated conditions. It is usually expressed as *time to failure* (in time units such as hours, cycles, miles, missions, etc.). In addition to random failures that occur throughout the life cycle, hardware engineers understand that components can suffer premature failure (often due to problems in design, testing and/or manufacturing) and/or may eventually wear out and require renewal (see expanded discussion in Section 1.6).

Software engineers uniformly do not have an analogous view of reliability. Webster defines reliable as "Giving the same result on successive trials". This definition, when extrapolated to include "forever", more closely resembles the view of reliability imposed on software engineers (by themselves and by the hardware community). In general, the reliability metric for software is used to describe the probability of the software operating in a given environment within the designed range of input without failure. Therefore, *software reliability* is a function of how well the software's purpose is delineated, built and tested; it is not a function of time. This concept is a fundamental issue that differentiates hardware and software reliability theory and assessment.

Additional differences between hardware and software include:

- The age of the software has nothing to do with its failure rate. If the software has worked in the past, it will work in the future, everything else remaining the same (i.e., no hardware, software or interface changes). Software does not rust or exhibit other wearout mechanisms like hardware does.
- The frequency of software use does not influence software reliability. The same software can be used over and over and, if it did not fail the first time, it will not fail any other time in identical usage (same range of inputs with no hardware, software or interface changes). In contrast, physical parts wear from usage, causing failures.
- Software does become obsolete, however, since user interface standards evolve and hardware environments become antiquated (see additional discussion in Section 1.6).
- With the exception of documentation and storage/transfer media, software, unlike hardware, cannot be held or touched. Typical methods of judging a hardware item include observing size and material composition, quality of assembly (form, fit and finish), and compliance with specification. For example, one can observe how well two gears mesh or if a transistor has sufficient current capacity for a circuit application. These physical concepts do not apply to software.
- Software cannot be judged prior to use by the same methods as hardware, i.e., there is no equivalent to incoming inspection.
- Software must be matched with hardware before it can ever be tested. If a failure occurs, the problem could be hardware, software, or some unintended interaction at the hardware/software interface.
- In general, hardware will either work or not in a given application. Software, aside from total failure, has varying degrees of success according to its complexity and functionality.
- Although not executable, documentation usually is considered an integral part of the software. Documentation which does not fully or accurately describe the operation can be considered to be just as much a failure as a software crash. When a user expects online help and does not get it (either because it is not activated or because what was provided was incorrect or incomplete), the software does not meet the user's expectation and, therefore, is not perfectly reliable. In contrast, documentation is usually not assessed when evaluating hardware reliability.

1.4 PHILOSOPHICAL AND CONCEPTUAL SIMILARITIES AND DIFFERENCES BETWEEN HARDWARE AND SOFTWARE QUALITY AND RELIABILITY

Quality Focus is the Same

One essential concept for both hardware and software is that the customer's perception of quality is extremely important. Quality is delivering what the customer wants or expects. Customers must be considered during the specification and design stages of development. Since various customer groups have conflicting interests and view quality and reliability differently, it is important to analyze the customer base.

For example, the organization funding a project is one customer, the user another. If they are different organizations, their expectations may be in conflict. Quality for the funding organization may be interpreted as "delivering on time and within budget" with "conformance to requirements" viewed as having less priority. In contrast, the customer who depends on the system's functionality to meet organizational needs is probably not as concerned with development schedule or cost. The pilot of a jet fighter expects the hardware and software to work perfectly regardless of whether the various sub-systems were delivered on time or within budget. Any failure, for any reason, is catastrophic. On the other hand, those accountable for verifying that the jet will not fail are very much interested in ensuring that both the hardware and software have been thoroughly tested and that the reliability assessment process is consistent with what has been used in other systems that have proved to be as reliable as predicted. The expectation is that quality consists of evidence that everything possible has been done to ensure failure-free operation, providing very high reliability.

Organizational Structure Often Causes Differences

The typical sequential organizational structure does not support significant cross communication between hardware and software specialists. An organization's internal communication gap can be assessed by considering the questions in Table 1-1. The answers help determine if the organizational structure creates two "separate worlds". If reliability is important and a communication gap exists, then the organization needs to break down the communication barriers and get all parts of the technical community to focus on a common purpose. Activities may involve awareness training, cross training, organizational restructuring, implementing/improving a metrics program, reengineering the overall system development processes as well as the sub-system (i.e., hardware and software) processes, or instituting a risk assessment/risk management program.

Lack of Methodology Knowledge Causes Differences

Reliability engineering encompasses a wide spectrum of methodology and analysis approaches that strive to systematically reduce, eliminate and/or control system failures which will adversely affect performance of a product. In cases where failures cannot be eliminated or controlled due to economic or design limitations, reliability engineering provides tools for determining the basis for risk assessment. Two examples from the many tools available are Fault Tree Analysis and Failure Mode, Effects and Criticality Analysis.

Table 1-1. Assessing The Organizational Communications Gap

- Is the software group a separate entity from the engineering group?
- Does the organization consider software development as an engineering discipline?
- What is the career path for hardware engineers? Software engineers? System engineers?
- What forums exist for interaction among software engineers, hardware engineers, system engineers and project managers?
- Who heads up system development? Hardware engineers? Software engineers? Others?
- Is there an expressed need for quantifying hardware reliability? Software reliability? System reliability?
- Who has defined the hardware reliability metric? Software reliability metric? System reliability metric? Are the definitions consistent?
- Who is responsible for assessing software reliability? Hardware reliability? System reliability?
- What metrics are in place for assessing hardware reliability? Software reliability? System reliability?
- What program is in place for testing hardware reliability? Software reliability? System reliability?
- Is there a common source of project data?

However, in the software environment these reliability engineering tools are generally not known or have not been used. Software engineers typically have no exposure to these techniques either in their formal training or in informal interactions with other software project personnel. The language of reliability engineering is often “foreign”; many reliability terms are not part of the software engineer’s basic technical vocabulary.

1.5 SOFTWARE RELIABILITY TERMINOLOGY

While hardware-focused reliability engineers have adopted a common set of concepts and terms with explicit meaning, the software community has not yet reached consensus and, hence, no universally adopted terminology set is in place. Many concepts, fundamental to the discussion and development of software reliability and quality, have several meanings. Worse, they are often used interchangeably!

For instance, software people often interchange “*defect*”, “*error*”, “*bug*”, “*fault*”, and “*failure*”. As an example, Capers Jones[3] defined the terms as:

- Error: A mistake made by a programmer or software team member that caused some problem to occur.
- Bug: An error or defect that finds its way into programs or systems.

[3] Jones (1994).

- Defect: A bug or problem which could cause a program to either fail or to produce incorrect results.
- Fault: One of the many nearly synonymous words for a bug or software defect. It is often defined as the manifestation of an error.

Some software specialists define a "*failure*" as any inappropriate operation of the software program while others separate "*faults*" and "*failures*" on a time dimension relative to when a defect is detected: "*faults*" are detected before software delivery while "*failures*" are detected after delivery. To the hardware community this appears to be an artificial distinction; yet it is important to be aware of the differentiation since both terms are used in actual practice. Software people talk about "*fault rate*" and "*failure rate*", with the latter term having a different meaning than that used with regard to hardware.

Robert Dunn[4] defines a software defect as "Either a fault or discrepancy between code and documentation that compromises testing or produces adverse effects in installation, modification, maintenance, or testing". In contrast, Putnam and Myers[5] define a defect as "A software fault that causes a deviation from the required output by more than a specified tolerance. Moreover, the software need produce correct outputs only for inputs within the limits that have been specified. It needs to produce correct outputs only within a specified exposure period". Since these definitions differ, a count of the number of defects will yield different results, and, hence, a different defect rate, depending on the counter's definition.

Dunn separates defects into three classes (he feels that it is fairly easy for experienced programmers to relate to each of these):

- Requirements Defects: Failure of software requirements to specify the environment in which the software will be used, or requirements documentation that does not reflect the design of the system in which the software will be employed.
- Design Defects: Failure of designs to satisfy requirements, or failure of design documentation to correctly describe the design.
- Code Defects: Failure of code to conform to software designs.

Typical requirements defects include indifference to the initial system state, incomplete system error analysis and allocation, missing functions, and unquantified throughput rates or necessary response times. The many kinds of design defects include misinterpretation of requirements specifications, inadequate memory and execution time reserves, incorrect analysis of computational error, and infinite loops. Possible code defects include unreachable statements, undefined variables, inconsistency with design, and mismatched procedure parameters.

---

[4] Dunn (1984), p. 6.

[5] Putnam and Myers (1992), p. 123.

Other software experts have different classifications. For example, Putnam and Myers define six classes of defects:

- Requirements Defects
- Design Defects
- Algorithmic Defects
- Interface Defects
- Performance Defects
- Documentation Defects[6]

The first three classes (first column) are essentially the same as those of Dunn, with the last three classes covering the rest of the software development cycle.

Capers Jones, a noted software metrics expert, uses six defect classes, as listed in Table 1-2.[7] Since most people do not comprehend the large number of defects that occur during software development, this table provides an illustration of the defects found during the development of one 200,000-line COBOL application. The resultant defect rate is 49.5 defects per thousand lines of code. Perhaps even more surprising is the distribution of the defects. Coding typically is expected to be the predominate defect category. Yet these results show that coding errors only account for about one in four defects. Requirements and design defects account for almost half of the total.

Table 1-2. Number of Software Defects by Classification Type

TYPE	Requirements	Design	Coding	Documentation	Administration	Bad Fixes	Total
NUMBER	1,800	2,600	2,400	1,400	110	1,600	9,910
PERCENT	18.2	26.2	24.2	14.1	1.1	16.1	100

1.6 LIFE CYCLE CONSIDERATIONS

Hardware Life Cycle

Hardware reliability often assumes that the *hazard rate* (i.e., failure rate per unit time, often shortened to the failure rate) follows the "bathtub" curve, illustrated in Figure 1-2. Failures occur throughout the item's life cycle; the hazard rate initially is decreasing, then is uniform, and finally is increasing.[8]

---

[6] Documentation defects are discrepancies between what the document says and what the code does, rather that edit problems.

[7] Cited in Putnam and Myers (1992), p. 124; previously appeared in Jones (1986).

[8] The implicit assumption is that the item is being operated within its specifications. If the item is being operated outside its specifications (overstressed), the failure is considered to be *induced*. Induced failures are usually considered to be outliers, or exceptions, and are not included in the observations which are used to formulate the prediction model or the reliability estimate.

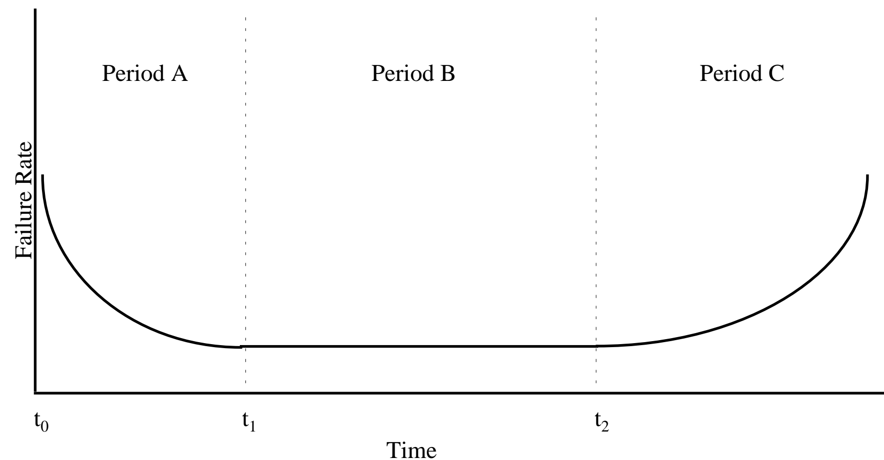


Figure 1-2. Bathtub Curve for Hardware Reliability

The time points on the plot are defined as follows:

- Time t_0 is the time the population of components is activated or put into service ("fielded" or "distributed"); usually this is after the completion of development and production (whose times are not shown on the figure; i.e., design, build and test times are not included). Failures occurring during Period A, from t_0 to t_1, are said to be due to *infant mortality.*
- Time t_1 is the time when nearly all items with manufacturing defects have failed and have been removed of the population. Failures occurring during Period B, from t_1 to t_2, are assumed to be *random*, i.e., not due to any specific factor. The user is confident that the component will remain in service during this period. The probability that the component will function until time t_2 is expressed as the probability of success or the *reliability*.
- Time t_2 is the end of the *useful life* when components begin to exhibit end-of-life failures. Those failures occurring during Period C, after t_2, are considered to be due to *wearout*.

In hardware, the number of infant mortality failures observed in the field can be reduced by testing (*screening*) the components or assemblies prior to distribution (i.e., in the bathtub curve, the height of the curve in Period A can be reduced; alternatively the length of time attributable to infant mortality (Period A) can be reduced, causing t_1 to be moved closer to t_0). In the case of electronic components, this screen consists of operating, or *burning in*, the component for a time usually less than or equal to t_1. In the case of mechanical components, the screen may also include visual inspection. In addition, a random sample of the items may be tested to demonstrate adherence to specification. These procedures may be performed by the item manufacturer prior to distribution to ensure that shipped components have few or no latent failures. Otherwise, the purchasing organization takes the responsibility for these activities.

When modeling the failure characteristics of a hardware item, the factors which contribute to the random failures must be investigated. The majority are due to two main sources:

- *Operating stress* is the level of stress applied to the item. The operating stress ratio is the level of stress applied relative to its rated specification. For example, a resistor rated to dissipate 0.5 watts when actually dissipating 0.4 watts is stressed at 80% of rated. Operating stresses are well defined and measurable.
- *Environmental stresses* are considered to be those due to the specific environment (temperature, humidity, vibration, etc.) that physically affect the operation of the item being observed. For example, an integrated circuit having a rated temperature range of 0° to 70°C that is being operated at 50°C is within operational environment specification. Environmental stresses also can be well defined and measurable.

When transient stresses occur in hardware, either in the operating stresses or the environmental stresses, failures may be induced which are observed to be random failures. For this reason, when observing failures and formulating modeling parameters, care must be taken to ensure accurate monitoring of all of the known stresses.

Software Life Cycle

The same "*bathtub*" curve for hardware reliability strictly does not apply to software since software does not typically wearout. However, if the hardware life cycle is likened to the software development through deployment cycle, the curve can be analogous for times up to t_2. For software,

- Time t_0 is the time when testing begins. Period A, from t_0 to t_1, is considered to be the *debug* phase. Coding errors (more specifically, errors found and corrected) or operation not in compliance with the requirements specification are identified and resolved. This is one key difference between hardware and software reliability. The "clock" is different. Development/test time is NOT included in the hardware reliability calculation but is included for software.
- Time t_1 is the initial *deployment* (distribution) time. Failures occurring during Period B, from t_1 to t_2, are found either by users or through post deployment testing. For these errors, work-arounds or subsequent releases typically are issued (but not necessarily in direct correspondence to each error reported).
- Time t_2 is the time when the software reaches the end of its useful life. Most errors reported during Period C, after t_2, reflect the inability of the software to meet the changing needs of the customer. This *obsolescence* period can be likened to the maintenance/ phase-out step of the software cycle. In this frame of reference, although the software is still functioning to its original specification and is not considered to have failed, that specification is no longer adequate to meet current needs. The software has reached the end of its useful life, much like the wearout of a hardware item. Failures reported during Period C may be the basis for generating the requirements for a new system.

Usually hardware upgrades occur during Period A, when initial failures often identify required changes. Software upgrades, on the other hand, occur in both Periods A and B. Thus, the Period B line is not really "flat" for software but contains many mini-cycles of Periods A and B: an upgrade occurs, most of the errors introduced during the upgrade are detected and removed, another upgrade occurs, etc. Hence Figure 1-3 might be a better representation of the software life cycle. Although the failure rate drops after each upgrade in Period B, it may not reach the initial level achieved at initial deployment, t_1. Since each upgrade represents a mini development cycle, modifications may introduce new defects in other parts of the software unrelated to the modification itself. Often an upgrade focuses on new requirements; its testing may not typically encompass the entire system. Additionally, the implementation of new requirements may inversely impact (or be in conflict with) the original design. The more upgrades that occur, the greater the likelihood that the overall system design will be compromised, increasing the potential for increased failure rate, and hence lower reliability. This scenario is now occurring in many legacy systems have recently entered Period C, triggering current reengineering efforts.

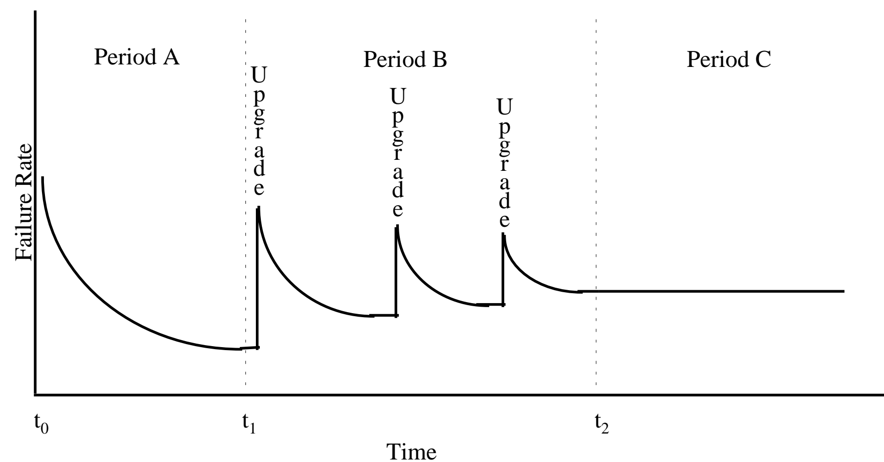


Figure 1-3. Revised Bathtub Curve for Software Reliability

In software, the screening concept is not applicable since all copies of the software are identical. Additionally, typically neither operating stresses nor operational environment stresses affect software reliability. The software program steps through the code without regard for these factors. Other quality characteristics, such as speed of execution, may be effected, however. The end user might consider a "slow" program as not meeting requirements. (This relates to the discussion in the next section on what constitutes a software failure.)

Table 1-3 summarizes the fundamental differences between hardware and software life cycles.

Table 1-3. Summary: Life Cycle Differences

LIFE CYCLE	PRE t_0	PERIOD A (t_0 to t_1)	PERIOD B (t_1 to t_2)	PERIOD C (Post t_2)
HARDWARE	Concept Definition Development Build Test	Deployment Infant Mortality Upgrade	Useful Life	Wearout
SOFTWARE	Concept Definition Development Build	Test Debug/Upgrade	Deployment Useful Life Debug/Upgrade	Obsolescence

1.7 SOFTWARE FAILURES MODES, EFFECTS AND CRITICALITY ISSUES

Reliability engineers often identify all possible failure modes, the resultant effects and the criticality of each. In considering software reliability, there are several aspects of this approach that need to be considered.

Failure Modes

The definition of what constitutes a failure in software is often open to debate. When a program "*crashes*", it has obviously failed due to an error, either in design, coding, testing, or exception handling.

> Software sometimes fails to perform as desired. These failures may be due to errors, ambiguities, oversights or misinterpretation of the specification that the software is supposed to satisfy, carelessness or incompetence in writing code, inadequate testing, incorrect or unexpected usage of the software or other unforeseen problems.[9]

However, the software may not crash and still fail. This can be due to a number of criteria which are not always well defined before development. Speed of execution, accuracy of the calculations and other criteria can all be significant factors when identifying lack of successful software operation. In addition, each of these criteria has a specific level of importance and is assessed on a specific scale.

When first using or evaluating a software program, a significant amount of time can be spent determining compliance with specification or applicability of an application. Depending on the person doing the evaluation, or the evaluation process scope and design, the evaluation may or may not fully exercise the program or accurately identify functionality that is unacceptable.

---

[9] Keiller and Miller (1991), p. 95-96.

Hardware/software interface problems can also occur, including failures in software due to hardware or communications environment modifications. Software errors can be introduced during software functional upgrades or during either scheduled or unscheduled maintenance.

In system testing, it is important to remain within the application environment and the range of intended use. If the evaluation tests the system (including both the hardware and software) beyond its specification limits, perceived lack of functionality will be identified which may result in unrealistically low reliability values. Too, if the user goes beyond the bounds of intended use, then the system may fail. This is in contrast to the concept of *accelerated testing* used in hardware evaluation where environmental stresses (typically outside the range of intended use) are applied to the hardware to induce failures (e.g., "shake and bake" tests involving extreme vibration and temperature cycling).

Software failures are usually considered relative to the application type and the severity of failure as evaluated by the specific end user. Consider the following two examples. One software program contains complex graphical user interfaces that map exactly to the customer's layout requirements; but this program crashes whenever a specific sequence of user inputs and events occurs. Another software program has layout flaws but it does not fail for any sequence of user triggered events. Which program is more reliable?

- Is an application reliable if it meets all specified requirements? Then the first is better.
- If failure is defined as any crash, then the second is more reliable; in fact, some would say it is perfectly reliable because it does not crash.

Are there trade-offs? How are they considered? That might be easier to answer if all requirements were of a binary nature (yes/no, present/absent), but they are not.

Another issue is that a part of the software (a "module") may function in one situation and not in another due to its interface with either the other software modules or the hardware.

Failure Effects

When hardware fails, the effects are usually immediately observed. Electronics overheat and smoke, indicating that the device is carrying too much current, or the system totally fails and shuts down. Mechanical hardware failure causes system shut down also but typically in a much more dramatic manner. In the most extreme cases, a catastrophic end effect, such as a plane crashing, is readily visible and attracts widespread attention.

When software fails, a dramatic effect, such as a plane crash, can also be observed. Often, however, the effect of a software failure is not immediately seen or may only cause inconvenience. A supermarket checkout system which incorrectly prices selected items may never be noticed, but a failure has still occurred. An Automatic Teller Machine (ATM) which does not allow user access is a nuisance which results in disgruntled customers. Both of these may be the result of catastrophic software failures, but, in reference to endangering human life, both are minor system failures.

These examples illustrate that is it important to distinguish between the software failure relative to the software's functioning as compared to the software failure relative to the total system's functioning. In the supermarket example, the software may have failed but the checkout continued while in the ATM example, the system did not operate.

Failure Criticality

Both hardware and software fall into two general categories based on the function performed: *mission critical* and *non-mission critical*. *Mission critical* encompasses all failures that are life threatening as well as failures that have catastrophic consequences to society. Table 1-4 identifies hardware failure severity levels with respect to both mission and operator. In hardware reliability improvement, usually only catastrophic and critical levels of severity are addressed.

Table 1-4. Hardware Failure Severity Levels[10]

TERM	DEFINITION
Catastrophic	A failure which may cause death or system loss (i.e., aircraft, tank, missile, ship, etc.).
Critical	A failure which may cause severe injury, major property damage, or major system damage which will result in mission loss.
Marginal	A failure which may cause minor injury, minor property damage, or minor system damage which will result in delay or loss of availability or mission degradation.
Minor (Negligible)	A failure not serious enough to cause injury, property damage, or system damage, but which will result in unscheduled maintenance or repair.

No similar set of criticality classifications has been adopted by the entire software community. Putnam and Myers have defined four classes of software defect severity and identify the corresponding user response as shown in Table 1-5. It is interesting to note that this classification is not with respect to operator or mission but views the software as an entity in itself. No application reference is included in the descriptions. Another interesting contrast is that any level of software defect can cause a catastrophic system failure. If the software crashes ("*Critical*"), miscomputes ("*Serious*"), provides a partly correct answer ("*Moderate*") or mis-displays the answer on the screen ("*Cosmetic*"), the resultant failure may be catastrophic, resulting in system and/or operator loss.

Table 1-5. Software Failure Severity Levels[11]

SEVERITY	DESCRIPTION	USER RESPONSE
Critical	Prevents further execution; nonrecoverable.	Must be fixed before program is used again.
Serious	Subsequent answers grossly wrong or performance substantially degraded.	User could continue operating only if he allows for the poor results the defect is causing. Should be fixed soon.
Moderate	Execution continues, but behavior only partially correct.	Should be fixed in this release.
Cosmetic	Tolerable or deferrable, such as errors in format of displays or printouts.	Should be fixed for appearance reasons, but fix may be delayed until convenient.

[10] MIL-STD-1629A (1980), p. 10.

[11] Putnam and Myers (1992), p. 125.

When considering the full software realm, only a small portion of software applications fall into the intuitive "mission critical" category; this includes embedded software and software involving massive data manipulation such as used in banking. This software typically is the focus of reliability concerns. A great deal of business software falls into the non-critical category (with the exception of financial and accounting software). While quality is the concern of all software customers, it is perceived differently by varying customer groups. Customers of non-mission critical software seldom address reliability directly. They tend to think of reliability and quality as being the same concept. They have no need to quantify reliability in terms of failure-free execution. For example, the customer of word processing software is usually less interested in reliability; rather the focus is compatibility with other products or flexibility of use. There is often a tradeoff because the product may not meet expectations for some functions and yet exceeds expectations for other functions. In many instances only a subset of the functionality is needed so the customer is only concerned with the performance (i.e., reliability) of that subset.

1.8 SYSTEM RELIABILITY ISSUES

Since a total integrated system contains both hardware and software subsystems, software reliability assessment is needed. Software developers are expected to develop a software subsystem reliability estimate for inclusion in the model for total system reliability. Hence, there is a need to assess the reliability of the software subsystem as a whole, and to assess the reliability of the individual software modules ("components"). Software modules, however, are often interdependent; how this affects the reliability often is not known. Additionally, there are some software reliability derivations that depend on the software's functional characteristics, adding a level of complexity since different parts of the software often have differing functionality.

Reliability engineers estimate system reliability based upon the reliability of the subsystem configuration (including redundancy considerations), parts and components. One of several published system reliability prediction models[12] is typically used, depending upon application type, model form and various assumptions. The specific model is essential in the development of prediction techniques, allocation procedures, design and analysis methodologies, test and demonstration procedures/control procedures, and so on, to ensure that a system can be designed and manufactured to perform as intended over its useful life.

Assuming that the individual hardware and software subsystem reliability values have been determined, an approach for combining these values into a final system reliability number has not been universally accepted.

- Does one separately determine the hardware and software subsystem reliability values and then combine them by multiplying the two values? This would assume independence of the two subsystems. But as stated earlier, this independence is in question since the software's operation is confounded with the hardware.

---

[12] Five of the often used system reliability prediction models are the ARP-1.0 Model (published by the Society of Automotive Engineers, used for electronics systems), the Bellcore Model (used for electronics), the CNET (National Center for Telecommunications Studies) Model (used for electronics), the DTRC-90/010 Model (developed by the U.S. Navy for mechanical systems) and models found in MIL-HDBK-217 (U.S. Air Force, used for electronics).

- Certainly the two subsystem reliability values are not additive -- a result greater than one could occur.
- Perhaps the system reliability model should contain a component term for software (see, for example, MIL-HDBK-217). The problem is how to define and assess this term so that it reflects the true relationship of the software reliability as imbedded in the hardware system.

Those who have specifically investigated software reliability have not resolved how to incorporate the software reliability value into the system reliability estimate.

1.9 SOFTWARE PRODUCT, PROCESS AND RESULTANT RELIABILITY METRICS

Software product and process metrics are used to assess and predict software product and process quality as well as to evaluate if improvement has occurred. Table 1-6 places these software metrics into four categories; it indicates a minimal set used in an ad hoc development environment as well as additional metrics that require additional refinement of the software development process. Note that usage typically drives the identification of metrics and the subsequent data collection processes. Determine first what must be measured and why. Then use that information to delineate the metrics and determine the data needed for those metrics. The chapters that follow address these metrics classifications. First the specifics of the software product metrics are presented. Then the metrics used to assess both project management and overall process capability are discussed. Finally, software product fault/failure metrics and reliability models are presented.

Table 1-6. Software Metrics Reflect Development Process Quality

METRIC LEVEL	MINIMUM METRICS	ADDITIONAL METRICS
DEVELOPMENT PROCESS QUALITY	Low Quality (Ad Hoc Process)	High Quality (Repeatable, Defined, Managed Process)
PRODUCT DEVELOPMENT METRICS	Size and/or functionality; measured by number of modules, lines of code (LOC), and/or function points (FP)	Application type, language, requirements, test coverage and complexity metrics
PROJECT MANAGEMENT METRICS	Schedule: Planned and actual staffing (calendar time) over project life Effort: Planned and actual person months required for project completion Project cost characteristics	Schedule and Effort: Broken down by life cycle phase and by personnel categories Productivity rates Project management metrics related to risk and configuration control
PROCESS METRICS		Capability Maturity Models Malcolm Baldrige National Quality Award
PRODUCT FAULT/FAILURE METRICS	Number of faults found during testing Failures or problems reported by the users	Faults and failures classified by life cycle phase and severity level Fault/failure rate/density

1.10 SUMMARY: HARDWARE AND SOFTWARE RELIABILITY DIFFERENCES

Table 1-7 summarizes some of the differences between hardware and software reliability that make drawing parallels between the two difficult.

Table 1-7. Comparison of Hardware/Software Characteristics[13]

CONCEPT	HARDWARE	SOFTWARE
Failure Cause	Failures can be caused by deficiencies in design, production, use, and maintenance.	Failures are primarily due to design faults. Repairs are made by removing the fault or modifying the design to make it robust against the condition that can trigger the failure.
Wearout	Failures can be due to wear or other energy-related phenomena. Sometimes warning is available before failure occurs (systems can become noisy indicating degradation & impending failure).	There is no wearout phenomenon. Software failures occur without warning, although very old code can exhibit an increasing failure rate as a function of errors induced into the code while making functional code upgrades.
Repairable System Concept	Repairs can be made that might make the equipment more reliable. This would be the case with preventive maintenance where a component is restored to a like-new condition.	The reliability of a dynamic software system can be enhanced by periodic restarting of the code. This cleans up the operating environment, empties queues and frees memory.
Time Dependency and Life Cycle	Reliability can depend upon both early life and wearout phenomena, that is, failure rates can be decreasing, constant, or increasing with respect to operating time.	Reliability is not time dependent. Reliability improvement over time may be affected but this is not an operational time relationship. Rather, it is a function of reliability growth of the code through detecting and correcting errors.
Time Dependency	Reliability is time related, with failures occurring as a function of operating (or storage) time.	Reliability is not time related. Failures occur when a program step or path that contains the fault is executed and triggers a failure. Once fixed, that failure cannot reoccur.
Environmental Factors	Reliability is related to environmental factors (i.e., temperature, vibration, humidity, etc.)	External environment does not affect reliability, except insofar as it might affect program inputs. Operational environment is a factor.
Reliability Prediction	Reliability can be predicted in theory from knowledge of design, usage and environmental stress factors.	Reliability cannot be predicted from any physical basis, since it entirely depends upon human factors in design. Some a priori approaches exist based upon the development process used and code size.
Redundancy	Reliability can usually be improved by redundancy. The successful use of redundancy presumes ready detection, isolation and switching of assets. It also presumes no common cause failure occurs.	Reliability cannot be improved by redundancy if the parallel program paths are identical, since if one path fails, the other will have the same error. It is possible to provide redundancy by having parallel paths, each with different programs written and checked by different teams (N block redundancy).
Interfaces	Hardware interfaces are visual, i.e., one can see a ten-pin connector.	Software interfaces (e.g., modules) are conceptual rather than visual.
Failure Rate Motivators	Failures can occur in components of a system in a pattern that is, to some extent, predictable from the stresses on the components and other factors. Reliability-critical lists are useful to identify high risk items.	Failures are not usually predictable from analyses of separate statements. Any statement may be in error. Reliability critical lists of failures are not appropriate. Interface code and exception handling code have been found more failure prone.
Standard Components	Hardware uses standard components as basic building block.	There are no standard parts in software, although there are standardized logic structures. Also, software reuse is being deployed, but on a limited basis.

[13] Keene (1994), p. 7.