Software Reliability Engineering: More Reliable Software Faster and Cheaper

2nd. Edition

John D. Musa

Software Reliability Engineering: More Reliable Software Faster and Cheaper

2nd. Edition

John D. Musa

j.musa@ieee.org
Website: http://members.aol.com/JohnDMusa

authorHOUSE

1663 LIBERTY DRIVE, SUITE 200
BLOOMINGTON, INDIANA 47403
(800) 839-8640
www.authorhouse.com

First published by AuthorHouse 08/11/04

ISBN: 1-4184-9387-2 (sc)
ISBN: 1-4184-9388-0 (dj)

Library of Congress Control Number: 2004096446

Printed in the United States of America
Bloomington, Indiana

This book is printed on acid-free paper.

Chapter 1 - Problem, Process, and Product

Are you short of time? Stressed out? Do you need more time for yourself and your family? If you are a software professional, an engineer who deals with software, or a software or engineering student who will be dealing with software in the future, this book is for you.

As a software development manager in 1973, I found myself struggling with customer demands that we produce software that was more reliable, built faster, and built cheaper. I didn't want to work any more hours; I was serious about my profession, but I also wanted to enjoy life. The only way out that I could see was to work smarter. Why not work out a better way to develop software? And why not use engineering background to develop a practice that would be quantitatively based?

I quickly found that many other people shared this problem and this vision. Together we have put together and are continuing to grow software reliability engineering. The purpose of this book is to teach it to you quickly and efficiently.

Software reliability engineering is a skill that can make you more competitive in this environment of globalization and outsourcing. This is true whether you currently work with software-based systems or are a university student learning to work with them for the future. You can't change the fact that customers want "More reliable software faster and cheaper." Your success in meeting these demands affects the market share and profitability of a product for your company, and hence your career. These demands conflict, causing risk and overwhelming pressure, and hence a strong need for a practice that can help you with them.

Software reliability engineering (SRE) is such a practice, one that is a standard, proven, widespread best practice that is widely applicable. It is low in cost, and its implementation has virtually no schedule impact. We will show what it is, and how it works. We will then outline the software reliability engineering process to give you a feel for the practice, using a single consistent example throughout the book. Finally, we will list some resources that will help you learn more about it.

I expect that the book will be of special value to you if you are a software developer, software tester, system engineer, system architect, acquirer of software, quality assurance engineer, reliability engineer, or development manager of a project that contains software; or, of course, a student preparing for one or more of these roles. It will help you as a text in learning the subject, a deskside companion as you start to apply software reliability engineering, and a reference in handling special situations you may encounter.

At this point, let me suggest that you turn to "How to use this book" so we can help you best match your needs to the structure of the book, which was specially conceived to provide this flexibility.

1.1 Core material

1.1.1 Problem and solution

1.1.1.1 What is the software practitioner's biggest problem?

A result of the growing maturity of software development is that it is no longer adequate that software simply works; it must now meet other customer-defined criteria. Software development is very competitive, and there are many competent software developers, spread throughout the world.

Surveys of users of software-based systems generally indicate users rate *on the average* the most important quality characteristics as: reliability / availability, rapid delivery, and low cost (in that order). In a particular situation, any of these major quality characteristics may predominate. These are primarily user-oriented rather than developer-oriented attributes. A large sampling of software people indicates that the most important software problem facing them is how to resolve conflicting demands that customers place on them for important quality characteristics of software-based systems.

Quantitative measures have existed for a long time for delivery time and cost, but the quantification of reliability (availability and reliability are closely related, so I will often use "reliability" as a shorthand term for both) for software has been more recent. It is very important, however, because the absence of a concrete measure for software reliability generally means that reliability will suffer when it competes for attention against schedule and cost. In fact, this absence may be the principal reason for the well known existence of reliability problems in many software products.

But these demands conflict. How? Well, for example, profitability is directly related to establishing and meeting precise objectives for reliability and availability. If the reliability / availability objective is set too high, delivery time may be too long. One of your competitors will beat you to market and your product will take heavy losses. If the reliability /

availability objective is set too low, your product may get a bad reputation. Sales may suffer, and the product will again take heavy losses. The risk can create overwhelming pressure on you.

Incidentally, the characteristic of "faster delivery" includes as a subset "with greater agility," which basically means "with more rapid response to changes in user needs or requirements."

In response to this problem of conflict, risk, and stress, software researchers and developers have given much attention to the mechanics of software development and they have built many tools to support the process. However, we have paid too little attention to the *engineering* of reliability in software-based products. By "engineering" software reliability, I mean developing a product in such a way that the product reaches the "market" at the right time, at an acceptable cost, and with satisfactory reliability. You will note that I put "market" in quotes; this is to convey a broad meaning beyond the world of commercial products. Even products built for the military and for governments have a "market" in the sense that product users have alternatives that they can and will choose if a product is too late, too costly, or too unreliable.

1.1.1.2 How does software reliability engineering approach it?

The traditional view of software development does not provide us with enough power to achieve the foregoing engineering goal. Thus, software reliability engineering was developed to help resolve the problem. It differs from other approaches by being primarily quantitative.

Since around 1973, we have gained much practical experience with software reliability engineering on actual projects. Some problems remain, but further progress in solving them is mostly dependent on confronting them in real project situations, gathering relevant data, and summarizing this data in a practically usable form. The measures are currently sufficiently accurate and useful that the benefits in using them exceed their costs.

A primary objective of software reliability engineering is to help the engineer, manager, or user of software learn to make more precise decisions. A strong secondary objective is to make everyone more concretely aware of software reliability by focusing attention on it. Better decisions can save money on a project or during the life cycle of a piece of software in many ways. Consider the following simple example for illustrative purposes: a 2-year software project has a system test period of about 6 months. The amount of testing is usually highly correlated with the reliability required, although it is not the only factor. Suppose you could establish that you need only 5 months of test to get the reliability that you need for the particular application. Then you might well be able to save 4 percent of the development cost of the project. Compare this with the cost of application of

software reliability engineering, which is typically 0.1 to 0.2 percent of project development cost. Software reliability engineering thus improves the productivity of software projects. The cost effectiveness of the methodology is high.

You add and integrate software reliability engineering (SRE) with other good processes and practices; you do not replace them. With software reliability engineering, you control the development process; it doesn't control you. It is not externally imposed; you use quantitative information to choose the most cost-effective software reliability strategies for your situation. Software reliability engineering is *not* about just making software more reliable and available. It *is* about achieving correct balance, based on customer needs, among reliability / availability, delivery time, and cost. This requires more talking with customers. It may be helpful to understand who the customer's customers and competitors are.

When software reliability engineering was being developed, it was noted that there has been a strong and often justified resistance to externally imposed processes. One of the reactions to the imposition has been approaches like extreme programming. Development should be driven by what customers need in the product, but it is easier to prescribe a process than it is to determine customer needs. Hence the problem. Of all the alternatives to externally imposed processes, software reliability engineering is particularly efficient, because it is based on quantitative information about customers. Thus, software reliability engineering and extreme programming (or more generally, agile development) can often enhance each other.

The product characteristics I have been describing, reliability, development time, and cost, are attributes of a more general characteristic, product quality. Product quality is the right balance among these characteristics. Getting a good balance means you must pick quantitative objectives for the three characteristics and measure the characteristics as development proceeds.

Projects, of course, have for some time been able to set objectives for delivery date and cost of products and measure progress toward these objectives. What has been lacking until recently has been the ability to do the same thing for reliability for software-based systems. Since the 1940's, we have been able to set reliability objectives and measure reliability for pure hardware systems. However, the proportion of such systems is now rapidly diminishing to near nonexistence. Hence the need for and the development of software reliability engineering.

Initial (and many present) approaches to measuring software reliability were based on attempting to count the faults (defects or bugs) found in a program. This approach is developer-oriented. In addition, what was often counted was in reality either failures (the occurrences of malfunction) or

corrections (for example, maintenance or correction reports), neither of which are equivalent to faults. Even if you correctly count faults found, they are not a good quality indicator. A large number of faults found can mean either poor program development or very good fault removal. Faults remaining is a better quality indicator, but reliability is even richer.

Reliability is user-oriented rather than developer-oriented. It relates to operation rather than design of the program, and hence it is dynamic rather than static [Saunier (1983)]. It takes account of the frequency with which problems occur. It relates directly to operational experience and the influence of faults on that experience. Hence, you can easily associate it with costs. It is more suitable for examining the significance of trends, for setting objectives, and for predicting when those objectives will be met. It permits one to analyze in common terms the effect on system reliability of both software and hardware, both of which are present in any real system. Thus, reliability measures are much more useful than fault measures.

This does not mean that some attention to faults is without value. A better understanding of faults and the human error process that causes them can lead to strategies to avoid, detect, remove, or compensate for them.

You can use software reliability engineering for any release of any software-based product, beginning at the start of any release cycle. Hence, you can easily handle legacy products. I use the term "software-based" to emphasize the fact that there are no pure software systems, so that hardware must always be addressed in your analysis. Before applying software reliability engineering to the system test of any product, you must first test (or verify in some other manner), and then integrate, the units or modules into complete functions that you can execute. Of course, you apply software reliability engineering throughout the project life cycle, not just the system test phase.

Software reliability engineering works by quantitatively characterizing two things, expected use and desired major quality characteristics. First, it increases effective resources and it delivers the desired functionality for the product under development much more efficiently by quantitatively characterizing the expected use of the product. It employs use and criticality information to:

1. Precisely focus resources (for example, review time, unit code and test time, test cases, and test time) on the most used and/or most critical functions

2. Maximize test effectiveness by making test highly representative of the field

We quantitatively characterize expected use by developing the operational profile.

You can also apply some of the increase in effective resources to increase agility (ability to respond to changing requirements), to introduce new technology and practice (which will later increase your efficiency further), or to achieve personnel objectives (increase personal growth and job satisfaction; reduce pace, working hours, and stress; etc.).

Critical functions are those that have great extra value when successful or great extra impact when failing. This value or impact can be with respect to human life, cost, or system capability. Critical functions may not be the most used ones. For example, critical safety systems in nuclear power plants are ideally never used but must be very reliable and available. Nuclear power plants have a critical operation SCRAM (Super Critical Reaction Ax Man). This name apparently comes from an early nuclear reactor in which the control rods that stopped the reaction were dropped into the reactor by chopping a rope that held them up. The action suggested by the name was perhaps well advised.

Second, software reliability engineering balances customer needs for the major quality characteristics of reliability, development time, and cost precisely and hence more effectively. To do so, it sets quantitative reliability and/or availability as well as delivery time and price objectives. We then apply our resources to maximize customer value by matching quality needs. We engineer software reliability strategies to meet these objectives. We also track reliability in test and use it as a release criterion.

With software reliability engineering you deliver "just enough" reliability and avoid both the excessive costs and development time involved in "playing it safe" and the risk of angry users and product disaster resulting from an unreliable product. You improve customer satisfaction. Engineering project software reliability strategies based on objectives may increase efficiency by 50 percent or more. The key is tailoring your activities to meet the "just right" objectives. For example, code reviews are often in the critical path and lengthen time to market. If the objective is only an intermediate degree of reliability, you may want to skip code reviews.

One example of setting right objectives is that of a computer terminal manufacturer who was losing market share for a particular firmware-driven terminal. Investigation indicated terminals were more reliable than customers needed and were willing to pay for. A competitor better met customer needs. This manufacturer regained market share by introducing a new line of less reliable terminals with a lower cost. The manufacturer used software reliability engineering to set and precisely meet this goal.

1.1.1.3 What's been the experience with SRE?

Software reliability engineering is a proven, standard, widespread best practice that is widely applicable. The cost of applying it is low and its impact on schedule is minor. The benefits are large. You can deploy software reliability engineering in stages to minimize impact on your organization.

The occasional problem that has been experienced in deployment has not been technical, but has been caused by resistance to change and/or lack of commitment to really applying software reliability engineering.

The AT&T International Definity® PBX [Lyu (1996), pp 167-8] project represents one application of software reliability engineering. Project software engineers applied it along with some related software technologies. In comparison with a previous software release that did not use these technologies, customer-found faults decreased by a factor of 10, resulting in significantly increased customer satisfaction. Consequently, sales increased by a factor of 10. There were reductions of a factor of two in system test interval and system test costs, 30 percent in total project development duration, and a factor of 10 in program maintenance costs [Abramson et al. (1992)].

Software reliability engineering is based on a solid body of theory [Musa, Iannino, and Okumoto 1987)] that includes operational profiles, random process software reliability models, statistical estimation, and sequential sampling theory. Software personnel have practiced software reliability engineering extensively over a period dating back to 1973 [Musa and Iannino (1991b)].

In one company, AT&T, it has been a Best Current Practice since May 1991 [Donnelly et al. (1996)]. Selection as an AT&T Best Current Practice was significant, as very high standards were imposed. First, you had to use the proposed practice on several (typically at least 8 to 10) projects and achieve significant, documented benefit to cost ratios, measured in financial terms (for software reliability engineering, they all were 12:1 or greater). You then developed a detailed description of the practice and how you used it on the projects, along with a business case for adopting it. Committees of experienced third and fourth level managers subjected the description and the case to a probing lengthy review. Typically, the review lasted several months, with detailed examinations being delegated to first level software managers and senior software developers. The review of the software reliability engineering Best Current Practice proposal involved more than 70 such people. In the case of software reliability engineering, comments requiring action before final review of the proposal exceeded 100. It required several staff-months of work to address these comments. Even

then, the software reliability engineering Best Current Practice was only one of five approved from the set of some 30 proposals made in 1991.

Software reliability engineering has been an IEEE standard since 1988. In addition, the American Institute of Aeronautics and Astronautics approved software reliability engineering as a standard in 1993, resulting in significant impact in the aerospace industry [AIAA (1992)]. The AIAA includes NASA and airframe manufacturers (Boeing, etc.)

A major handbook publisher (McGraw-Hill) sponsored a *Handbook of Software Reliability Engineering* in 1996, further evidence of the field's importance [Lyu (1996)].

Organizations that have used software reliability engineering include Alcatel, AT&T, Bellcore, CNES (France), ENEA (Italy), Ericsson Telecom (Sweden), France Telecom, Hewlett-Packard, Hitachi (Japan) IBM, Lockheed-Martin, Lucent Technologies, Microsoft, Mitre, Motorola, NASA's Jet Propulsion Laboratory, NASA's Space Shuttle project, Nortel, North Carolina State University, Raytheon, Saab Military Aircraft (Sweden), Tandem Computers, the US Air Force, and the US Marine Corps, to name just a few. There is a selection of over 65 papers and articles by users of software reliability engineering, describing their experience with it, in Appendix F.

Tierney (1997) reported the results of a survey taken in late 1997 that showed that at that time Microsoft had applied software reliability engineering in 50 percent of its software development groups. This included projects such as Windows NT and Word. The benefits they observed were increased test coverage, improved estimates of amount of test required, useful metrics that helped them establish ship criteria, and improved specification reviews.

AT&T's Operations Technology Center in the Network and Computing Services Division has used software reliability engineering as part of its standard software development process for several years. This process was included in ISO certification. The Operations Technology Center was a primary software development organization for the AT&T business unit that won the Malcolm Baldrige National Quality Award in 1994. At that time, it had the highest percentage of projects using software reliability engineering in AT&T. Another interesting observation is that four of the first five software winners of the AT&T Bell Laboratories President's Quality Award used software reliability engineering.

As further witness to the value of software reliability engineering, growth of this field is very strong. The technical community grew from around 40 people at the founding of the first professional organization in 1990 to several thousands by 2004. There is an active annual International

Symposium on Software Reliability Engineering (ISSRE), which has grown steadily, even through downturns in the software industry.

Experience with the application of software reliability engineering indicates that the total cost is low. There is an investment cost in training and planning of no more than three equivalent staff days per person in an organization, plus an operating cost. The principal operating cost in applying software reliability engineering is that for developing the operational profiles (defined in Chapter 2) required. The cost of developing the operational profiles for a system varies with its size and the accuracy required, but an average effort is one to two staff months for systems in the range of hundreds of thousands of source instructions. Updating an operational profile for subsequent releases is a much smaller task. Very approximately, you can compute (from fitted data) the operating cost of software reliability engineering over the project life cycle as a percent of project software development costs, using

$$P = 10\,S^{-0.74} \tag{1.1}$$

where P is the percent and S is the project size in staff years. For small projects, the total effort is usually no more than one staff month.

The impact of software reliability engineering on project schedules is negligible. Most software reliability engineering activities involve only small effort that can parallel other software development work. The only significant activity in the critical path is two days of training. Since software reliability engineering helps you work more efficiently, the overall effect is to save time.

Software reliability engineering has many additional advantages. It is very customer- and user-oriented. It involves substantial direct beneficial interaction of project personnel with customers and users. This enhances a supplier's image, improving customer and user satisfaction and reducing the risk of angry users.

It deals with the complete product, even though it focuses on the software part. The focus is on software because software reliability has been neglected for so long.

It is highly correlated with attaining Levels 3, 4, and 5 of the Software Engineering Institute Capability Maturity Model. Software reliability engineering users note that it is unique, powerful, thorough, methodical, focused, and requirements-based. It takes a full-life-cycle, proactive view. It involves and improves cooperation among marketers, product managers, system engineers, process engineers, system architects, developers, testers, managers, field service personnel, and users (or their representatives).

Software reliability engineering improves project planning and it has a particularly strong influence on improving the quality of requirements.

Software reliability engineering increases respect for testing. Testers are brought into the development process earlier, often on the system engineering team, with the result that you are more likely to avoid building systems that cause testing problems.

Software reliability engineering is widely applicable. Technically speaking, you can apply software reliability engineering to any software-based product, beginning at start of any release cycle. It can actually be easier to implement software reliability engineering on a legacy product rather than a new product, as there is more information available on the legacy system to develop the operational profile.

In general, software reliability engineering is economically practical for systems that are considerably larger than units (blocks of code commonly developed by one programmer). The complete software reliability engineering process may be impractical for small components such as units that involve less than two staff months of effort, unless they are used in a large number of products. However, for single products, it may still be worthwhile to implement software reliability engineering in abbreviated form.

Software reliability engineering is independent of development technology and platform. It works well with such practices as agile development, extreme programming, and object-oriented development and it can readily be applied to web platforms. It requires no changes in architecture, design, or code, but it may suggest changes that would be beneficial.

Some people wrongly think that software reliability engineering is not applicable to their development process because they use terms that are different than those used by software reliability engineering, although the basic concepts are actually the same. I have not yet found a case where software reliability engineering could not be applied, although there have been special situations where adaptations had to be made. Hence, doubters should first make sure they understand software reliability engineering terminology and look for adaptations for special circumstances

When applying software reliability engineering, testing of the *units* that make up a system must be completed first.

1.1.2 The software reliability engineering process

The software reliability engineering process is designed for both legacy and new products. The software reliability engineering process proper consists of six activities. They are: defining the product, implementing operational profiles, engineering the "just right" reliability, preparing for

test, executing test, and guiding test. They are illustrated in Figure 1.1, with the project phases in which you customarily perform them shown at the bottom of the figure. Implementing operational profiles and engineering the "just right" reliability can be performed in parallel. The outputs of implementing operational profiles are used in preparing and executing tests. Note that "executing test" and "guiding test" occur simultaneously and are closely linked.

We will focus on the software reliability engineering process in this book in order to teach it, but remember that your task in learning it is to connect and integrate it with your overall software development process. You will see that you tailor it somewhat to the particular product you are developing.

I discuss each of these activities at length in a chapter of this book (Chapters 1 to 6), but I will provide a summary here.

The process diagram, for simplicity, shows only the *predominant* order of workflow, indicated by the arrows, in the software reliability engineering process. In actuality, the tasks frequently iterate and feed back to earlier tasks in a manner analogous to the spiral (as contrasted to the waterfall) model of the overall software development process. Just as some requirements and architecture changes may follow test in the software development process, changes in the engineering the "just right" reliability activity may follow executing test and guiding test in the software reliability engineering process.

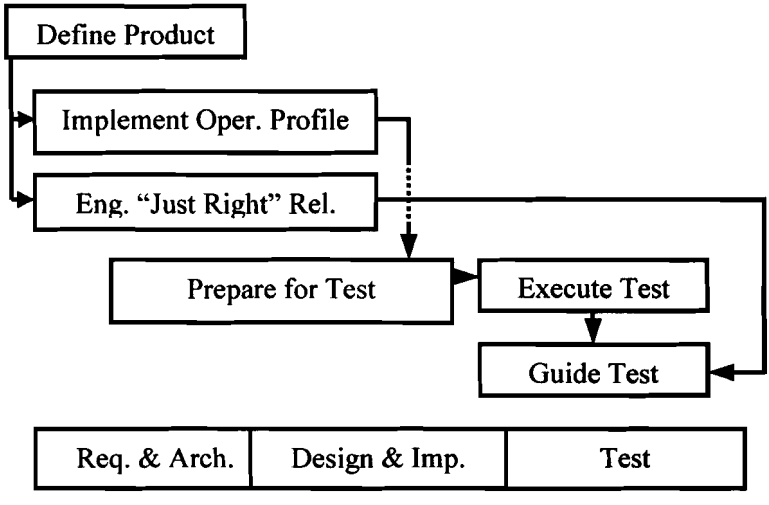

Figure 1.1 Process diagram

The Post Delivery and Maintenance life-cycle phase, which follows Test, is not shown in Figure 1.1. During this phase, you can determine the reliability actually achieved and the operational profile really experienced. This information affects the engineering of "just right" reliability and the implementation of the operational profile for the next release. It can result in reengineering of the product and the development process.

The first activity is to define the product. This includes establishing who the supplier, customers, and users are and listing the associated systems. The associated systems include the base product, variations, and other systems specially related to the product that are tested separately. An example of associated systems is the base product Fone Follower and its variation Fone Follower Japan. Some associated systems may also be delivered or to some extent be developed separately from other associated systems, but separate testing is the most important defining characteristic.

Next is implementing operational profiles. An operational profile is a complete set of operations (major system logical tasks) of a system with their probabilities of occurrence. We will both develop operational profiles and apply them. They will be applied to increase the efficiency of development (including determining requirements and architecture) of all developed software and test of all software. You may even use the operational profile to allocate resources among modules to cut schedules and costs.

The third activity is engineering the "just right" reliability. The principal subactivities in doing this are:

1. Defining failure in project-specific terms with examples

2. Choosing a common reference unit for all failure intensities (for example, failures with respect to hours, calls, or pages)

3. Setting a system failure intensity objective (FIO) for each associated system

4. Finding a developed software failure intensity objective for any software you develop, plus choosing among software reliability strategies (such as fault tolerance and requirements review) to meet the developed software failure intensity objective and schedule objectives with the lowest development cost

Obviously, we must have a clear idea of what reliability and the closely related availability are in order to engineer them.

The standard definition of *reliability* for software [Musa, Iannino, and Okumoto (1987)] is the probability that a system or a capability of a system will continue to function without failure for a specified period in a specified environment. "Failure" means the program in its functioning has not met user requirements in some way. "Not functioning to meet user requirements" is really a very broad definition. Thus, reliability incorporates many of the properties that can be associated with execution of the program. For example, it includes correctness, safety, and the operational aspects of usability and user-friendliness. Note that safety is actually a specialized subcategory of software reliability. Reliability does not include portability, modifiability, or understandability of documentation.

The period may be specified in natural or time units. Thus, we use a definition that is compatible with that used for hardware reliability, though the mechanisms of failure may be different. The compatibility enables us to work with systems that are composed of both software and hardware components. A *natural unit* is a unit other than time that is related to the amount of processing performed by software-based product, such as runs, pages of output, transactions, telephone calls, jobs, semiconductor wafers, queries, or API calls. Other possible natural units include data base accesses, active user hours, and packets. Customers generally prefer natural units. *Failure intensity*, an alternative way of expressing reliability, is stated in failures per natural or time unit.

Availability is the average (over time) probability that a system or a capability of a system is currently functional in a specified environment. Availability depends on the probability of failure and the length of downtime when failure occurs. We usually define *software* availability as the expected fraction of operating time during which a software component or system is functioning acceptably. Assume that the program is operational and that we are not modifying it with new features or repairs. Then it has a constant failure intensity and a constant availability.

We can compute availability for software as we do for hardware. It is the ratio of uptime to the sum of uptime plus downtime, as the time interval over which the measurement is made approaches infinity. The downtime for a given interval is the product of the length of the interval, the failure intensity, and the mean time to repair (MTTR). Usually the failure intensity applied here is a figure computed for serious failures and not those that involve only minor degradation of the system. It is generally not practical to hold up operation of the system while performing fault determination and correction in the field. Therefore, we ordinarily determine MTTR as the average time required to restore the data for a program, reload the program, and resume execution. If we wish to determine the availability of a system

containing both hardware and software components, we find the MTTR as the maximum of the hardware repair and software restoration times.

Then you prepare for test, which first involves specifying new test cases for the new operations of the base product and variations for the current release, based on the operational profile. Then you prepare test procedures to execute load tests, based on the operational profiles. Test procedures are the controllers for your load tests.

The fifth activity is executing test, during which you determine and allocate test time, invoke test, and identify system failures along with when they occurred. Testing in software reliability engineering is "black box" while other testing such as unit testing is "white box."

The final activity is guiding test, processing failure data gathered in test for various purposes. First, you track the reliability growth or reduction in failure intensity of developed software of the base product and its variations as you remove the faults that are causing failures. Then you certify reliability of the base product and variations that customers will acceptance test, plus the reliability of supersystems, or larger systems of which your product is a part. Certification is go-no go testing. All of these subactivities help guide product release.

The software reliability engineering process is not complete when you ship a product. You need to collect certain field data to use in engineering succeeding releases and other products. In many cases, we can collect the data easily and inexpensively by building recording and reporting routines into the product. In this situation, we collect data from all field sites. For data that requires manual collection, take a small random sample of field sites.

We collect data on failure intensity and on customer satisfaction with the major quality characteristics of the product, and use this information in setting the failure intensity objective for the next release. We also measure operational profiles in the field and use this information to correct the operational profiles we estimated. Finally, we collect information that will let us refine the process of choosing software reliability strategies in future projects.

Experience indicates that testers should participate in the first two activities, "implement operational profiles" and "engineer the 'just right' reliability," in partnership with system engineers. We originally thought that these activities should be assigned solely to system engineers and system architects. However, this did not work well in practice. Testers depend on these activities and are hence more strongly motivated than system engineers and system architects to insure their successful completion. We resolved the problem when we made testers part of the system engineering and system architecture teams.

This approach also had unexpected side benefits. Testers had much more contact with product users, which was very valuable in knowing what system behavior would be unacceptable and how unacceptable it would be, and in understanding how users would employ the product. System engineers and system architects obtained a greater appreciation of testing and of where requirements and design needed to be made less ambiguous and more precise, so that test planning and test case and test procedure design could proceed. System testers made valuable contributions to architecture reviews, often pointing out important capabilities that were missing.

The architecture phase includes design of fault tolerant features. This design is driven by the failure intensity reduction objective that is determined by software reliability engineering in the choice of reliability strategies.

Let's consider an illustration that we will apply throughout this book. My objective is to help make the process of applying software reliability engineering more concrete. A deliberate choice was made to present one unified example, as the author received strong participant feedback in the classes he taught to the effect that multiple examples were confusing and time-consuming (you had to learn about each and follow it). Workshop sections will also refer to the same illustration to refresh your memory.

I draw the illustration from an actual project, with the details modified for the purposes of simplicity and protecting any proprietary information. I selected this particular example because it deals with telephone service and hence can be understood by almost anyone. Also, it was small (total staff less than 10, with the extra effort required by software reliability engineering about 1 staff month). It in no way implies that software reliability engineering is limited in application to telecommunications or to small projects. In fact, software reliability engineering has been applied in my knowledge to systems ranging from 5,000 to 25,000,000 lines of source code. Applications outside of telecommunications include (at least) medical imaging, knowledge-based systems, a wide area network-based education system, a capital management and accounting system, a compiler, terminal firmware, instrument firmware, military systems, and space systems.

Fone Follower was a system that enabled you to forward incoming telephone calls (fax or voice) anywhere in the world, including to cellular or other portable phones. You, as a subscriber, called Fone Follower and entered the telephone numbers (forwardees) to which you wanted calls forwarded as a function of time.

Calls from the telephone network that would normally be routed to your telephone were sent to Fone Follower. It forwarded both fax and voice calls in accordance with the program you entered. If there was no response to a

voice call, you were paged if you had pager service. If there was still no response or if you didn't have pager service, the voice calls were forwarded to your voice mail.

Fone Follower had a base product for the U.S., Fone Follower, and a variation for Japan, Fone Follower Japan. This book follows Fone Follower through two releases. The second release, Fone Follower 2 (Fone Follower2), added two new operations, labeled Y and Z, to the original Fone Follower.

Fone Follower used a vendor-supplied operating system whose reliability was not known when we adopted it. Subscribers viewed service as the combination of standard telephone service with the call forwarding provided by Fone Follower. Fone Follower had 60,000 lines of developed executable source code. Fone Follower Japan had 20,000 lines of developed executable source code not found in Fone Follower. The average load was 100,000 calls / hr.

1.1.3 Defining the product

In order to define a product, you first need to establish who the supplier is and who the customers and users are. This may sound like a trivial question, but it often is not, in this complex global marketplace. For example, your organization may only be part of a corporation or alliance of corporations that combine multiple systems into a final system. The supplier is the organization that is applying software reliability engineering, and you may choose to consider this as your organization or your corporation or work at both levels in parallel. If you take your organization as the supplier, your customers and users may be other organizations in your corporation. If you take your corporation as the supplier, your customers and users will probably be entities outside your corporation.

I distinguish customers from users in the sense that customers make the purchase decision for a product and users put it to work. These can be but are not necessarily the same entities. You may have just one customer who is also the user, but usually there will be multiple customers and users, in which case you will want customer profiles and user profiles that indicate customers and users by types, with percentages for each type. The percentages may simply represent the numbers in each type, but most commonly, they represent proportions of revenues or profits coming from each type.

The other main subactivity in defining a product is to list its associated systems. You will always have a base product, and in a simple case, that is all you will have. The base product for Fone Follower was Fone Follower (U.S.).

However, many products will have variations. A *variation* of a product is a product that performs the same general function as the base product, but that differs sufficiently such that it is tested separately from it. It may differ in functionality (by having different operations) or in implementation (where it has the same functions but has different code due to different hardware, platforms, or environments). A different hardware configuration of the product (one that accomplishes the same functions but has different code) is a variation. Fone Follower Japan was a functional variation. A Macintosh version of a functionally identical word processor would be an implementation variation. Another example of implementation variations is different printer drivers for different platforms such as Windows XP, Macintosh, Unix, etc. International products may have to operate with different interfaces in different countries, and thus have different variations. Functional variations can be implemented by using software common to all variations plus modules that implement the functions that are particular to a variation.

Several factors can help you establish whether you have a separate variation, though any particular factor may not be a determining one:

1. Are the users substantially different?
2. Is the environment in which the product operates different?
3. Are there some functions that are substantially different?
4. Are you operating on a different platform?
5. Is the hardware you interface with different?
6. Is the design concept different?
7. Is the software developed by different people?
8. Is the testing separate?
9. Is administration of changes separate?

The more positive answers you have to these questions, the more likely you have a separate variation.

The other type of associated system you may have is the supersystem. The base product and each variation can have one or more supersystems. A *supersystem* is a set of interactionally-complex systems that include the base product or a variation plus independent systems, where users judge the

product by the set's reliability or availability. The interactions between the base product or variation and independent systems are so complex that they may not be completely known. Hence, you can't simulate the interactions by testing the base product or variation plus drivers.

Fone Follower had two supersystems, the supersystem of the Fone Follower base product with the US telecommunications network, and the supersystem of the Fone Follower Japan variation with the Japanese telecommunications network.

Be careful not to make the common mistake of considering the supersystem as just the set of external independent systems. It includes the base product or variation. Usually if a supersystem exists for the base product, similar supersystems exist for all variations. You often have supersystems in the case of packaged software. They also commonly occur with printers and peripherals.

Some supersystems can't practically be tested in system test (usually because of the economics of providing the independent systems for the system test laboratory), but they can be tested in beta test. Beta test involves direct execution in the field environment, identifying failures and applying failure data to certify the supersystems. Beta test does not require developing operational profiles, test cases, or test procedures.

There is, of course, a limit to how many associated systems you will identify. Each additional associated system you identify and test involves extra cost and possible schedule delays, even though you can conduct some multiple tests in parallel. Costs include developing one or more additional operational profile(s), developing test cases and procedures, executing the tests, and in some cases, additional development costs. You should list an additional associated system only when the benefits of separate test outweigh the extra time and costs involved. The benefits are the reductions in risk of customer dissatisfaction or schedule delays (both leading to lost market share) and reduction in risk of extra failure resolution costs. Limit the number by eliminating those with infrequent use. Projects often follow the process of brainstorming to produce many potential associated systems, followed by a reality check to limit the list to the most important ones. This approach is very common in software reliability engineering.

The costs of separately testing an additional system may be reduced if a system has an operational architecture (its components correspond to operations or groups of operations), because it's relatively simple to obtain operational profiles for the components and hence relatively easy to separately test the components.

1.2 Exercises

1.1 We have a text processing software product, developed totally within our software house. It must work with operating systems 1 and 2 and printer systems (hardware and software) A, B, C, and D. What systems should we test?

1.2 We have an object library of five objects that we use, respectively, in 20, 12, 4, 2, and 1 systems. Which objects would you select for separate test?

1.3 Our product uses a database that our subcontractor will deliver two weeks before the start of system test. Should we certification test it?

1.4 An expert system (software) assists open heart surgeons by continuously analyzing patient's echocardiogram and electrocardiogram, alerting surgeon to potential problems (e.g., 40 percent chance of fibrillation), and recommending responses. How reliable should this software be?

1.5 Can you have too much reliability?

1.6 What rank order of the quality characteristics is most important for your application?

1.7 Can you give an example of a natural unit from your project?

1.8 Can you give an example of associated systems from your project?

1.9 Can you give an example of a variation from your project?

1.10 Can you give an example of a supersystem from your project?

1.3 Workshop

1.3.1 General instructions for all workshops

The purpose of the workshops is to guide you step by step in applying software reliability engineering to a *simplified* product that is as close to the one you work on as possible. We anticipate that you will bring the special engineering judgment that you have developed with experience in your knowledge domain and for your product to the workshops and synthesize this with the new practice of software reliability engineering. For the above reason, we deliberately do not use a canned example, because then you just rehearse the manipulations of software reliability engineering and do not experience the more difficult challenge (which occurs for all new practices)

of applying it to reality. Our goal for you is to use the results of the workshops as a base and guide for a full implementation of software reliability engineering on your project.

There is one caution: we sometimes restrict the number of entities (associated systems, operations, etc.) or simplify situations that we deal with in the workshops so that you can complete the workshops quickly. When you deal with a real project, you cannot do this, so don't be led astray by following the workshops blindly.

If you decide to work in a group, note that group size is important; the optimum is between 6 and 12. The absolute minimum size for an effective group is 4. The absolute maximum size for an effective group is 15. Small groups provide insufficient diversity for a good learning experience; large groups do not permit each individual to participate. If it is not possible for you to work in a group, you may wish to have someone with software reliability engineering expertise look over and critique your workshop templates, just as you would have someone critique your writing.

You must now pick a specific software-based product and a release of that product to work with. Identify a software-based product that best meets the following criteria:

1. The product is real (not "constructed")

2. You have a reasonable understanding of the product.

3. The product is moderate in size (perhaps 50,000 to 500,000 lines of code) and complexity since the total workshop time is planned for about 6 hours. This may mean that you should take only part of a large, complex project, perhaps a subsystem or a feature. On the other hand, a product that is too small and simple will not exercise all the things you have learned.

4. Work on a specific release of the product is about to start or is not yet out of the requirements phase, so that you can apply the results of your set of workshops to it.

As far as I know, there are no projects to which you can't apply software reliability engineering for technical reasons.

At the start of each workshop, there is a list of inputs and outputs expected for that workshop. When you start the workshop, check to see if you have all the inputs you will need. If not, plan to get them as action items. Make the best estimate or assumption you can for now so that you

can proceed. At the end of the workshop, make sure you have generated all the outputs required that are applicable to your project.

The illustrations of the tasks for each step use Fone Follower wherever possible. Please remember that this illustration is a simple one for teaching purposes; your project may well be more complex.

Templates are provided for each workshop so that you can record your workshop *results* in a systematic way. You should do this as soon as you determine the results of each step. Action items can be designated temporarily by an arrow "→" in the left margin but you will probably want to transfer them later on to the integrated list provided at the end of this first workshop. Once you have filled in the templates, you will have in one place both a comprehensive book on software reliability engineering *and* an example of its application to one of your own projects.

At several points, you will create lists (of associated systems, operations, etc.). Start by brainstorming freely, considering a wide range of possibilities. Then apply judgment to reduce the length of the list so it will be manageable in the actual deployment of software reliability engineering. You may want to further limit the list to make the workshop an efficient learning experience; instead of repeating a procedure unnecessarily, you can scale up its use in deployment.

Actively question what you are doing as you go. If you think you might do something differently a second time, note it as an action item for investigation and learning.

Put particular emphasis on the early steps of the software reliability engineering process, just as you would with the software development process. "Up front" investment will pay off downstream. However, since the software reliability engineering process follows a spiral model, some actions will later be changed. If you are missing information needed for a decision, list obtaining that information as an action item. Make the best estimate or assumption you can and proceed. If time does not permit you to fully complete the activities outlined here (this is expected for many systems), simplify by considering one part of your project or by making assumptions; record the work not done as action items. When you are dealing with numbers, pick simple ones (even if approximate) when you are doing hand calculations. But make sure they are as realistic as possible, so that you obtain quality insights about your product. If a step is not applicable to your product, do not execute it. However, you should review it for learning purposes.

Please note the following common traps to avoid:

1. Misunderstanding of definitions and concepts. Since terms may vary from company to company or even within a company, make sure you

understand correctly what the terms in the template mean. Refer to the glossary (Appendix B).

2. Too many items of anything (for example: associated systems, operations).

3. Getting bogged down in excessive details, with the result that learning to deal with difficult major issues suffers. One common example is spending excessive time on precise calculation.

4. Not making assumptions and assigning action items when information is missing.

5. Defining operations, runs, and test cases that take too long to execute. Break these down, with a test procedure exercising overall control over them.

6. Working with steps in the process out of sequence. This can lead to neglecting something important and causing serious trouble later on. Those who have done this often end up having to redo all the steps anyway. However, if you carefully consider a step and conclude it is not applicable to your project, you may skip it without a problem.

1.3.2 Defining the product workshop

Inputs:

1. High level product requirements and architecture (from project information)

2. Information on how users judge product: Do they view it as being coupled to independent systems? (from project information)

Outputs:

1. Summary description of product, focused on its functions (to Implementing operational profiles and Engineering "just right" reliability workshops)

2. List of associated systems (to all workshops)

STEP 1 OF 2. Define your product. Determine what the product is, who the supplier is, and who the customers and users are. This may be nontrivial when you are an organizational unit within a large company. Establish the principal functions of the product to a sufficient extent that the user's view of the product is clear. Record a summary description of the product. One of the principal functions of this summary description is to convey the nature of the product quickly and concisely to others who may look at your workshop notes. You do not need to describe detailed functions or to set reliability and/or availability objectives at this point.

Illustration - Fone Follower: Summary description of product:

A subscriber calls Fone Follower and enters the phone numbers to which calls are to be forwarded as a function of time. Incoming calls (voice or fax) from the network to the subscriber are forwarded as per the program. Incomplete voice calls go to a pager (if the subscriber has paging service) and then voice mail. Fone Follower uses a vendor-supplied operating system of unknown reliability. Subscribers view the service as the combination of standard telephone service with call forwarding.

The supplier is a major telecommunications systems developer. The customers are telecommunications operating companies, and they sell Fone Follower service to a wide range of businesses and individuals

STEP 2 OF 2. List the associated systems of your product. In addition to the base product, these may include variations of the base product, supersystems of the base product, and supersystems of variations. You may find it useful to draw a block diagram of the product and its environment; it will often be particularly helpful in identifying supersystems. The size of the test effort will increase with the number of systems; hence, you should add to this list with care. For purposes of the workshop, limit the associated systems you will work with to the base product, one of the base product's supersystems, one variation, and one of that variation's supersystems

Illustration - Fone Follower:

base product: Fone Follower
variation: Fone Follower Japan
supersystem of base product: US telephone network and Fone Follower
supersystem of variation: Japanese network and Fone Follower Japan

Action Item List for Workshops

Item no.	Action required	Person resp.	Date

Action Item List for Workshops

Item no.	Action required	Person resp.	Date

1.4 Special situations

1.4.1 Testing acquired software

At one time, it appeared that as a routine matter we should separately acceptance test all *acquired software,* the software that we as a supplier did not develop ourselves. Acquired software includes "off the shelf" or packaged software, software that we reuse from another product, component, or object library; and subcontracted software. Note that you should consider software that you develop within your company but not under the managerial control of your project, as being subcontracted.

Unfortunately, the testing of acquired software has not proved out to be useful in general practice, although it may still be beneficial in special situations. The idea was to check any acquired software whose reliability was unknown or questionable early in product development. The goal was early detection of unreliability, so that you could take appropriate actions to prevent schedule delays or cost overruns. Unfortunately, most development schedules turned out to be too challenging for this approach to have time to work.

There are two cases, however, in which such test may be useful. The first is products with very high reliability requirements and plenty of time for development. In this case, your supplier can deliver the acquired software to you for acceptance test well before your system integration starts, leaving sufficient time to resolve any reliability deficiency. The other case is the pretesting and certifying of components that a supplier is making available for use in products. Because of the difficulties involved in testing acquired software soon enough to be useful, you should try to reduce the number of such tests by selecting acquired software with a reputation for satisfactory reliability.

The key concept is that the system should be "assurable," which means that it can be checked for reliability and that correction can be made in a timely fashion if it is deficient. Determining which acquired software could cause problems and is assurable requires engineering and management judgment of the risks involved. It is generally economic to test such a system only if it is a major component or a smaller component used in many products, such as an object.

Usually the system must be available at least 3 months and perhaps 6 to 9 months ahead of the start of system test for there to be enough time for standalone test and corrective action. There are several possibilities for correction: returning the component to the supplier, substituting a competitor's component, or developing a workaround. If the system cannot meet the foregoing conditions, you do not have assurable acquired software.

Note that test of assurable acquired software occurs before system test of the overall product. Hence, it does not require allocation of test cases or test time from that period.

You will usually test such a system only for the first release, because ordinarily it will not change appreciably between releases. The exception, of course, is when you substitute a different acquired system, for example, from a competing supplier, or when the functions the acquired software must perform change substantially. The testing that should be done is certification testing (see Section 6.1.2). The definition of a failure for an acquired software component is behavior that would cause a failure in the base product or variation of your product, as you have defined it.

For very high reliability products, the best approach is to apply software reliability engineering to acquired software jointly, involving yourself as customer and your supplier as developer. You would then perform an acceptance test after the system is delivered to you, rejecting it for rework if it does not pass the test. If your supplier does not wish to participate in software reliability engineering or the acquired software has already been developed, then you simply apply the acceptance test.

If you know that a component has sufficient reliability for your product from previous experience, it is not necessary to test it separately. If you are not sure of the component but have a good relationship with the supplier, you may decide to have them perform reliability growth test, under surveillance, and skip the acceptance testing, saving time and cost.

Note that size is an important factor in deciding whether to test a component. If the component is a major one, testing it separately can be cost effective. If you have small acquired software components (including objects), separate testing will be cost effective only if they are used in many systems.

There appears to be great potential in the application of software reliability engineering to pretesting of components that are being made available for use in products. Examples include "off the shelf" software and the object libraries needed for object-oriented development, since you can expect to use the components in many systems. In fact, the further growth and use of off the shelf components and object-oriented development may depend on the marriage of these technologies to software reliability engineering. Object-oriented concepts have made better modularization possible, but we often are not realizing the promise and benefits of reuse because developers (probably rightly!) have enormous resistance to using objects of undetermined reliability.

Although it may not be cost effective to test more than a few systems separately, it is not as expensive and hence it may be practical to determine failure intensities for a larger number of separate components that are tested

together. This is because you only have to classify failures by components and measure processing done for those components (either by counting natural or time units expended). However, because of the errors in estimating failure intensity from small failure samples, making multiple failure intensity estimates may not be practical.

1.5 Background

In this section, I will provide further general background for the practice of software reliability engineering. I will explore in greater depth some of the concepts related to software reliability. Then I will compare software reliability with hardware reliability. For further background, see Musa (1999) and Musa, Iannino, and Okumoto (1987).

1.5.1 Learning reliability concepts

We can define reliability quantities with respect to natural units or time units. We are currently concerned with two kinds of time. The *execution time* for a program is the time that is actually spent by a processor in executing the instructions of that program. The second kind of time, *operating time*, also called *calendar time*, is the familiar garden variety of time that we normally experience. Execution time is important, because it is now generally accepted that models based on execution time are superior. However, quantities must ultimately be related back to operating time to be meaningful to many engineers or managers. If computer utilization, which is the fraction of time the program is executing, is constant, operating time will be proportional to execution time. As an example of these two types of time, consider a word processing system serving a secretary. In one week, there may be 40 hr of time during which the system is running. There might be 2 hr of execution time for the word processing program itself.

Weekly average computer utilization, although often relatively constant in the field, tends to increase over a system test period. You can describe the relationship of operating time to execution time during system test using the calendar time model [Musa, Iannino, and Okumoto (1987)].

There are four general ways of characterizing failure occurrences in time:

1. Time of failure
2. Time interval between failures
3. Cumulative failures experienced up to a given time
4. Failures experienced in a time interval

We illustrate these four ways in Tables 1.1 and 1.2.

Table 1.1 Time-based failure specification

Failure number	Failure time (sec)	Failure interval (sec)
1	10	10
2	19	9
3	32	13
4	43	11
5	58	15
6	70	12
7	88	18
8	103	15
9	125	22
10	150	25
11	169	19
12	199	30
13	231	32
14	256	25
15	296	40

All the foregoing four quantities are random variables. By "random," we mean that we do not know the values of the variables with certainty. This frequently happens when so many factors (many unknown) affect the values that it is not practical to predict them. There are many possible values, each associated with a probability of occurrence. For example, we don't really know when the next failure will occur. If we did, we would try to prevent or avoid it. We only know a set of possible times of failure occurrence and the probability of each. The probability of each time of occurrence is the fraction of cases for which it occurs.

Note that "random" does not carry any connotation of true irrationality or unpredictability, as some mistakenly assume. "Random" means "unpredictable" only in the sense that the exact value is not known. However, the average value and some sense of the dispersion *are* known. "Random" does not mean "unaffected by other variables." Although failure occurrence is random, it is decidedly affected by such factors as test strategy and program use. Nor does "random" imply any specific probability distribution, which some mistakenly assume as "uniform."

Table 1.2 Failure-based failure specification

Time (sec)	Cumulative failures	Failures in interval
30	2	2
60	5	3
90	7	2
120	8	1
150	10	2
180	11	1
210	12	1
240	13	1
270	14	1

There are at least two principal reasons for this randomness. First, the commission of errors by programmers, and hence the introduction of faults, is a very complex, unpredictable process. Hence, the locations of faults within the program are unknown. Second, the conditions of execution of a program are generally unpredictable. For example, with a telephone switching system, how do you know what type of call will be made next? In addition, the relationship between program function requested and code path executed, although theoretically determinable, may not be so in practice because it is so complex. Further, the code path executed and the occurrence or nonoccurrence of failures may be influenced by environmental factors that are not predictable. Examples of such factors are interaction with other functions, traffic level, and data corruption. Since failures are dependent on the presence of a fault in the code *and* its execution in the context of certain machine states, a third complicating element is introduced that argues for the randomness of the failure process.

Table 1.3 illustrates a typical probability distribution of failures that occur within a time period of execution. Each possible value of the random variable of number of failures is given along with its associated probability. The probabilities, of course, add to 1. Note that here the random variable is discrete, as the number of failures must be an integer. We can also have continuous random variables, such as time, which can take on any value.

Note that the most probable number of failures is 2 (probability 0.22). The mean or average number of failures can be computed. You multiply each possible value by the probability it can occur and add all the products, as shown. The mean is 3.04 failures.

You can view a random process as a set of random variables, each corresponding to a point in time. The random variables themselves can form

a discrete or continuous range. They can have various probability distributions. One common probability distribution for failure random processes is Poisson.

Table 1.3 Typical probability distribution of failures

Value of random variable (failures in time period)	Probability	Product of value and probability
0	0.10	0
1	0.18	0.18
2	0.22	0.44
3	0.16	0.48
4	0.11	0.44
5	0.08	0.40
6	0.05	0.30
7	0.04	0.28
8	0.03	0.24
9	0.02	0.18
10	0.01	0.1
Mean failures	----	3.04

The other principal characteristic of a random process is the form of the variation of the process in time. We can have a discrete or continuous time random process. We will look at the time variation from two different viewpoints, the mean value function and the failure intensity function. The *mean value function* represents the average cumulative failures associated with each time point. The *failure intensity function* is the rate of change of the mean value function or the number of failures per unit time. For example, you might say 0.01 failure / hr or 1 failure / 100 hr. Strictly speaking, for a continuous time random process, the failure intensity is the derivative of the mean value function with respect to time, and is an instantaneous value.

Each alternative form of expressing reliability, failure intensity or reliability proper, has its advantages. The failure intensity form is easier to state, as you only have to give one number. Further, it is easier to understand for those without probability and statistics background. However, when you require proper operation of a system for some time duration to accomplish some function, reliability is often better. An example would be a space flight

to the moon. Figure 1.2 shows how failure intensity and reliability typically vary during a test period, as faults are removed. The time for the test period will typically represent hundreds or thousands of runs. Note that we define failure intensity, just as we do reliability, with respect to a specified environment.

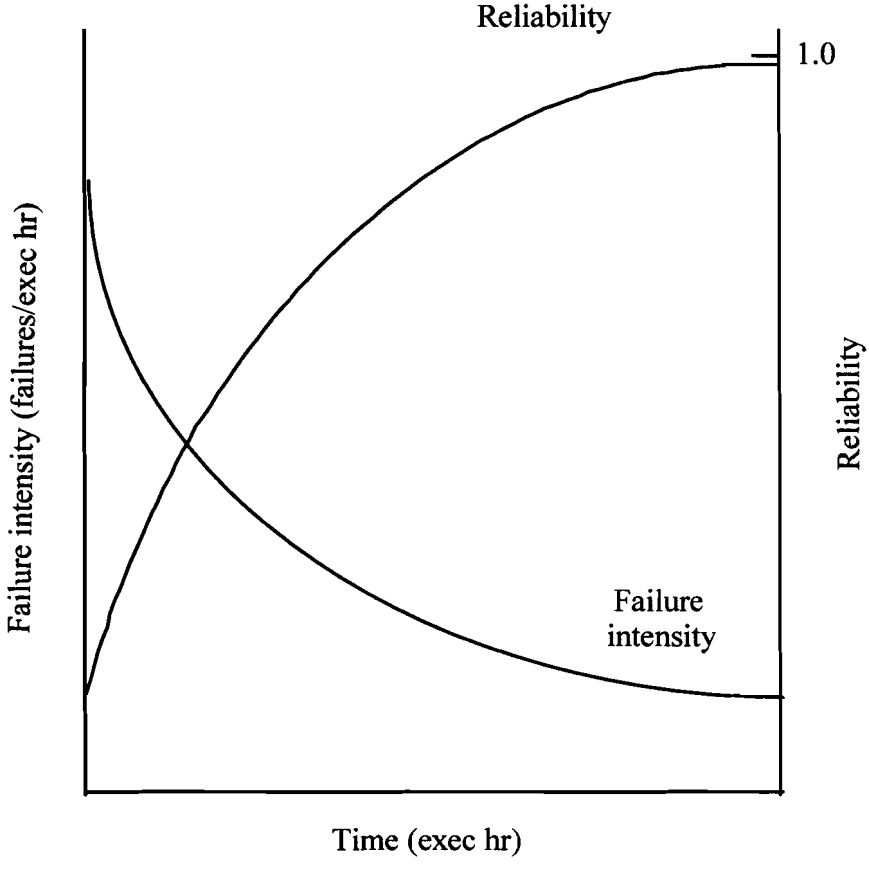

Figure 1.2 Reliability and failure intensity

As we remove faults (for example, during a reliability growth test phase), failure intensity tends to drop and reliability tends to increase. When we introduce new features or make design changes, we usually introduce new faults. The result is a step increase in failure intensity and a step decrease in reliability. If we introduce new features and fixes at the same time, there can be a step increase or decrease in failure intensity, depending on whether the faults introduced by new features or the fault removals predominate. There will be a corresponding decrease or increase in

reliability. If a system is stable (that is, the code is unchanging), both failure intensity and reliability tend to be constant. This would be the situation when we release a program to the field, making no changes in code and no repairs.

One uses the term *mean time to failure (MTTF)* in the hardware reliability field and to a decreasing extent in software reliability. It is the average value of the next failure interval. The use of mean time to failure is attractive, in that "larger" indicates "better." However, there are some cases in software reliability in which mean time to failure is undefined from a rigorous mathematical viewpoint. Failure intensity is preferred because it always exists. In addition, failure intensities are simple to work with because they combine additively. In an approximate nonrigorous sense, the two are inverses of each other. The hardware reliability field uses the term *mean time between failures (MTBF)* when repair or replacement is occurring. It is the sum of mean time to failure and mean time to repair (MTTR).

Table 1.4 illustrates an example of the random process of failures in simplified fashion, showing the mean value function of the cumulative number of failures experienced at two different time points. The time points are $t_A = 1$ hr and $t_B = 5$ hr. A *nonhomogeneous* random process is one whose probability distribution varies with time. Most failure processes during reliability growth test fit this situation.

The two most important factors affecting failure behavior are:

1. The number of faults in the software being executed

2. The operational profile

The number of faults in the software is the difference between the number introduced and the number removed during its lifetime.

Programmers introduce faults when they develop code. They may introduce faults during original design or when they are adding new features, making design changes, or repairing faults that have been identified. The term *developed code* refers to instructions that you create or modify as a supplier in your organization. In general, only code that is new or modified results in fault introduction. Code that is *reused* to perform the same functions that it performs in another application does not usually introduce any appreciable number of faults, except possibly in the interfaces. It generally has been thoroughly debugged in the previous application. Note that the process of fault removal often introduces some new faults because it involves modification or writing of new code. Hopefully (if you have a programming staff of reasonable competence!), the new faults entered are fewer than the faults being removed.

Table 1.4 Probability distributions at times t_A and t_B

Value of random variable (failures in time period)	Probability with elapsed time $t_A = 1$ hr	Probability with elapsed time $t_B = 5$ hr
0	0.10	0.01
1	0.18	0.02
2	0.22	0.03
3	0.16	0.04
4	0.11	0.05
5	0.08	0.07
6	0.05	0.09
7	0.04	0.12
8	0.03	0.16
9	0.02	0.13
10	0.01	0.10
11	0	0.07
12	0	0.05
13	0	0.03
14	0	0.02
15	0	0.01
Mean failures	3.04	7.77

Fault removal obviously can't occur unless you have some means of detecting the fault in the first place. Thus, fault removal resulting from execution depends on the occurrence of an associated failure. Occurrence depends both on the length of time for which the software has been executing and on the operational profile. When you execute different operations, you encounter different faults under different conditions and the failures that are exhibited tend to be different - thus the effect of the operational profile. For example, you can expect that the software reliability of an electronic telephone switching system operating in a business district will be different from that of one operating in a residential area.

We can often find faults without execution. They may be found through inspection, compiler diagnostics, design or code reviews, or code reading.

Fault removal is also dependent on the efficiency with which faults are found and removed. For example, a failure resolution team may remove 95 faults for every 100 failures reported. The efficiency depends on factors

such as how well the circumstances surrounding the failure are documented and the degree of structuring and clarity of the program.

We have seen that the failure process depends on the *system* being built, the nature of the development process for a particular *project*, and the *use* of the system. Since the system and the project are so closely associated, they are often used interchangeably in naming a particular failure process.

The following relationships apply to reliability generally (hardware or software). Reliability, denoted *R(t)*, is related to failure probability *F(t)* by:

$$R(t) = 1 - F(t) \tag{1.2}$$

The failure probability is the probability that the time of failure is less than or equal to *t*. If *F(t)* is differentiable, then the failure density *f(t)* is the first derivative of *F(t)* with respect to time *t*. The hazard rate *z(t)* is the conditional failure density, given that no failure has occurred in the interval between *0* and *t*. It is given by

$$z(t) = \frac{f(t)}{R(t)} \tag{1.3}$$

and it is also related to the reliability by

$$R(t) = \exp\left[-\int_0^t z(x)dx\right] \tag{1.4}$$

In software reliability engineering, the hazard rate in the field is piecewise constant, discontinuities occurring only at new releases. The hazard rate is equal to the failure intensity λ. Hence, from Equation (1.4) you can derive the relationships between reliability and failure intensity given in Appendix C.

The mean time to failure (MTTF) Θ is related to the reliability by

$$\Theta = \int_0^\infty R(x)dx \tag{1.5}$$

where the integration is performed with respect to the operating time of the system.

1.5.2 Software reliability and hardware reliability

The field of hardware reliability has been established for some time. Hence, you might ask how software reliability relates to it. In reality, the

division between hardware and software reliability is somewhat artificial. Both may be defined in the same way. Therefore, you may combine hardware and software component reliabilities to get system reliability.

Both hardware and software reliability depend on the environment. The source of failures in software is design faults, while the principal source in hardware has generally been physical deterioration. However, the concepts and theories developed for software reliability could really be applied to any design activity, including hardware design. Once a software (design) defect is properly fixed, it is in general fixed for all time. Failure usually occurs only when a program (design) is exposed to an environment that it was not developed or tested for. Although manufacturing can affect the quality of physical components, the replication process for software (design) is trivial and can be performed to very high standards of quality.

Engineers have not applied the "design reliability" concept to hardware to any extent. The probability of failure due to wear and other physical causes has usually been much greater than that due to an unrecognized design problem. It was possible to keep hardware design failures low because hardware was generally less complex logically than software. Hardware design failures had to be kept low because retrofitting of manufactured items in the field was very expensive. The emphasis in hardware reliability may be starting to change now, however. Awareness of the work that is going on in software reliability, plus a growing realization of the importance of design faults, may be having an effect. This growing awareness is strengthened by the parallels that people are starting to draw between software engineering and chip design. Also, practitioners are realizing that firmware is really software.

Since introduction and removal of design faults occur during software development and test, you can expect software reliability to vary during these periods. Hardware reliability may change during certain periods, such as initial burn-in or the end of useful life. However, it has a much greater tendency than software toward a constant value.

Despite the foregoing differences, we have developed software reliability theory in a way that is compatible with hardware reliability theory. Thus, you can compute system reliability figures using standard hardware combinatorial techniques [Shooman (1986) and Lloyd and Lipow (1977)]. In summary, hardware and software reliability share many similarities and some differences. One must not err on the side of assuming that software always presents unique problems, but one must also be careful not to carry analogies too far.

1.6 Frequently asked questions

1.6.1 Problem and solution

1.6.1.1 What is the software practitioner's biggest problem?

1. Why is software reliability so important in the area of software quality?

   ANSWER: Software reliability is a very specific and measurable property in the larger concept of software quality, which can be rather fuzzy. It is probably the most inclusive and important property, since it deals with freedom from departures from user-specified behavior during execution. In other words, the software executes in the way the user wants it to. Other aspects of quality not incorporated in reliability (for example, maintainability) will be strictly secondary if the program does not operate satisfactorily.

2. We resolve every failure that occurs in test. Why then do we need to estimate software reliability?

   ANSWER: Because otherwise, you will have no idea of how much additional testing you should do.

3. Why do we need to measure software reliability? Wouldn't it be more effective to establish the best software development process we can and apply it vigorously?

   ANSWER: The essential question is, "What do you mean by 'best'?" Someone must define "best," preferably the users of the ultimate software-based product. A definition without measurement is very imprecise. We need to know the *level* of reliability that users require. And note that "highest level" may be wrong, because the resulting development time and cost may be unacceptable. Finally, we have no satisfactory way of determining the activities that should be incorporated into a process unless we can measure their effectiveness in yielding desired

product characteristics such as reliability, development time, and cost.

4. Does the Capability Maturity Model (CMM) sponsored by the Software Engineering Institute require measurement of reliability of software-based systems?

   ANSWER: Yes.

5. My company has recently merged with another and we are now reorganizing to maximize our efficiency. Can software reliability engineering be of any help to us?

   ANSWER: Yes. Developing an operational profile for the entire business as a system will provide valuable insight into your company's business and logical ways in how it may be organized.

6. We are undertaking a process improvement program. Can software reliability engineering help?

   ANSWER: Yes. First you can use software reliability engineering to estimate the relative impact of different failures by multiplying the failure intensity of each by the per failure impact. You can then determine which failures are worth further investigation and in what order. The purpose of the investigation is to discover the root cause of the failure and what process improvement would be most cost effective in eliminating the root cause.

   Different process improvements can be evaluated, using software reliability engineering, for their impact on failure intensity. Then the change in failure intensity and the cost of the process change can be used to determine the most cost effective change.

7. How can you anticipate software reliability problems?

   ANSWER: There are many indicators. Instability of requirements or a poor development process are two of them. Once you start system test, and

are tracking the FI / FIO ratio, an FI / FIO ratio at the start of system test that is much higher than you planned when you chose your software reliability strategies is a forewarning of trouble.

8. You have noted that, in general, the relative importance of significant quality attributes is reliability, schedule, cost. Are there any major departures from this order?

   ANSWER: Yes. Government and military applications often rank cost ahead of schedule.

9. Shouldn't we speak of meeting user expectations rather than user needs?

   ANSWER: Not in most cases. Meeting user expectations may lead to the greatest satisfaction in the short term, but ultimately winning market share will depend on meeting user needs. The exception might be products where fashion is very important, such as computer games, where short term satisfaction rules, because people won't use the product over the long term.

10. What does the purchaser of software need to know about software reliability engineering? Is there anything different about the way a purchaser would apply the software reliability engineering process?

    ANSWER: The purchaser participates in setting reliability objectives, developing operational profiles, identifying failures, and certifying software, and therefore needs the skills associated with these activities. The purchaser applies the software reliability engineering activities just described in the same way the software vendor does, but with the perspective of different interests.

11. Should customers be totally uninterested in the estimated number of faults that remain in a system that is delivered to them?

ANSWER: No, this is as extreme as *only* being interested in the number of faults. The number of faults remaining indicates the effort and cost involved in elevating the software to a higher level of reliability and has some relationship to how long this should take. *They should have little interest in how many faults the developer has removed to date, however.* Their prime interest should be in the failure intensity, because that will indicate the level of their operational problems and costs.

1.6.1.2 How does software reliability engineering approach it?

1. Doesn't the copying of hardware reliability by software reliability result in an inappropriate technology?

   ANSWER: Software reliability was *not* copied from hardware reliability. It was developed with full cognizance of the differences. However, we deliberately defined software reliability to be mathematically compatible so that the reliability of systems with both components could be evaluated. We studied but rejected other approaches such as fault counting because of their incompatibility, among other reasons. An incompatible definition would not be practically useful, since real systems always have components of both types.

2. Can software reliability engineering help us with Total Quality Management, as characterized by the Baldrige award?

   ANSWER: Software reliability engineering is integrally connected and is in fact a *keystone* to Total Quality Management. It provides a user-oriented metric that is highly correlated with customer satisfaction. It is a keystone in the sense that you cannot manage quality without a user-oriented metric of system reliability, and you cannot determine reliability of software-based systems without a software reliability measure. Software reliability engineering is

associated with most of the major categories in the Baldrige award scoring system.

3. Can software reliability concepts contribute to root cause analysis (the determination of the ultimate causes of faults, done with the objective of improving the software engineering process that generated the faults)?

   ANSWER: Yes. The operational profile (defined in Chapter 2) highlights those functions that execute frequently. Failure reports related to these functions or to failures with high severity of impact should receive priority in root cause analysis studies. Then quality improvement actions undertaken on the basis of root cause analysis will address those causes first that will have the greatest effect on customer perception of quality.

4. Where can software reliability engineering make its greatest contributions in improving the software development process?

   ANSWER: In speeding up the development and test of function (and at the same time reducing cost) for a product while maintaining the necessary reliability.

5. My company is evolving from a developer of software to a company that primarily integrates software developed by others. How will this affect the software reliability engineering we do?

   ANSWER: You will continue to have a strong need for software reliability engineering because you must continue to ensure the reliability of the final product and achieve the right balance among reliability, development time, and cost. In fact, you probably will have a greater need for software reliability engineering because you will want to certification test many of the components being delivered to you. However, you will find less need for reliability growth testing.

6. Sometimes I hear of new software reliability engineering ideas that I do not find in this book. Why?

   ANSWER: I have deliberately limited this book to ideas that have been proved in practice, unless I note otherwise. Some of the ideas you have heard about may indeed be promising, and some of them may be proven in practice in the years to come. What you see here is the current state of the practice.

7. How can software reliability engineering improve testability?

   ANSWER: Operational profiles and failure intensity objectives focus the attention of developers on testing. Software reliability engineering provides a structured, efficient way to test. Both factors enhance testability.

8. How can software reliability engineering decrease time to software deployment?

   ANSWER: Software reliability engineering makes software development more efficient by focusing resources on the functions that are most used and most critical. This efficiency will yield the same reliability in reduced deployment time.

9. How does software reliability engineering affect serviceability (the degree to which software faults can be found or countered in the field)? Note that serviceability is related to recoverability and diagnosability but not to fault tolerance.

   ANSWER: The features that make software more serviceable are operations that record status, diagnose and display deviant behavior, etc. Proper development of an operational profile ensures that the need for and expected amount of use of these operations is noted. Software reliability engineering should then result in the right level of reliability for them.

10. Can you give a simple example to demonstrate why focusing effort with the operational profile improves efficiency?

ANSWER: Yes. Suppose you have a system with operations A and B. Operation A executes 90 percent of the time and B, 10 percent. Assume that each operation has 10 faults. Further assume that you have 10 hours of test and debugging effort available and finding and removing each fault requires one hour of effort.

If you do not have operational profile information, you will devote 5 hours of effort to each operation, reducing the number of faults in each to 5. Since A runs 90 percent of the time and B runs 10 percent of the time, you will experience 0.9 (5) + .01 (5) = 5 faults.

With the operational profile, you will devote 9 hours of effort to operation A, reducing its faults to 1. You will devote 1 hour of effort to operation B, reducing its faults to 9. Now you will experience 0.9 (1) + 0.1 (9) = 1.8 faults.

With the same effort, you have reduced faults encountered, and thus the failure intensity, by almost a factor of 3.

11. The term "software reliability" is disturbing. Software doesn't wear out like hardware, so how can you apply a hardware concept?

ANSWER: First, the term "software reliability" is in general use, like it or not. However, I have deliberately decided to promote the usage further to emphasize compatibility of software and hardware reliability calculations and their joint use in computing system reliability. It is true that software doesn't wear out in the same sense as hardware, although it tends to degrade with time as its functions evolve and as it is maintained. "Software reliability" is really a measure of confidence in the design. You could estimate design confidence for hardware systems. You usually don't, because wear-out phenomena tend to swamp out design defects for most hardware systems.

12. Aren't we stretching things when we say that software failure is random? Hardware really does seem to fail randomly but software is the result of an intellectual process that can be controlled.

    ANSWER: The randomness of hardware failure is no more "real" than that of software. The concept of randomness is used to model behavior that is affected by so many factors that a deterministic model is impractical. We may think that we control intellectual processes and hence their products, but this is illusory. Many defects (for example, memory lapses) occur in intellectual processes. A wide range of factors cause these defects, so the whole is best modeled by a random process.

13. How can you talk of a reliability for software? It isn't the same situation as we have for hardware. You let me pick the software tests and establish their sequence and I'll create any reliability estimate you want. For example, if you take a run that executes without failure and repeat it endlessly, you will get a reliability that approaches one.

    ANSWER: Manipulation of the reliability estimate is possible. You must know at least one run that fails and one that succeeds. You then select them in the proportions you desire. However, we aren't really interested in behavior of the program for an artificial set of executions. We are concerned with the typical environment of runs (usually randomly sequenced) that will normally be encountered. Manipulation of tests is also possible for hardware and just as unreasonable as for software. For example, you can obtain a very low reliability for most automobile tires by running them incessantly over a set of spikes.

14. How can you view the use of a system as being random? If people learn the weaknesses of a system, they will avoid them.

    ANSWER: Suppose people do learn the weaknesses of a system and avoid the faulty operations rather than request the software designer to

correct them. This acceptance is equivalent to a redefinition of the system requirements. Incidentally, there is no reason why use of the remaining operations should not be regarded as a random process.

15. How can software reliability vary with time, since it doesn't wear out?

ANSWER: Note that software reliability varies with *execution time*. As failures occur and faults underlying them are uncovered and corrected, reliability will increase. Any variation of software reliability with time occurs solely because time and execution time have some relationship.

16. Is software reliability engineering requirements-based?

ANSWER: Yes. This does not mean that we ignore useful information we may have about the design or the history of the project.

17. What are some of the common misconceptions about software reliability engineering?

ANSWER: The most common misconceptions are:

a. That it is primarily concerned with software reliability models and prediction

b. That it incorrectly copies hardware reliability theory

c. That it only deals with faults or bugs

d. That it doesn't concern itself with representative testing

e. That testing ultrareliable software is hopeless

18. Why is software reliability so different from hardware reliability? I know software is quite different from hardware, but why can't software failure data be treated like hardware failure data?

    ANSWER: Software and hardware reliability do have the same definition: probability of failure-free operation for a specified time. The principal difference is that software reliability is more likely to change with time than hardware reliability, either as faults are removed during reliability growth test or as new code (which may contain additional faults) is added. Hence the need for models to characterize this change. If software is stable (not evolving and not being debugged), the software failure data can be treated like hardware failure data.

19. Which is more appropriate, a run-oriented view of software reliability or a time-oriented one?

    ANSWER: Since all runs have a frequency of occurrence and a duration during execution, the two views are equivalent.

20. Do the environmental factors that influence hardware reliability affect software reliability?

    ANSWER: In general, no. Hardware reliability is mostly affected by physical variables like temperature and mechanical stress. Software reliability is mostly affected by how the software is used.

21. How will the move to network computing (i.e., applets, Java, client-server networks, etc.) affect software reliability engineering?

    ANSWER: It should not affect basic theory and it probably will not affect the basic methods used in practice. However, we can expect to see more emphasis on small software components used in a wide variety of applications within a domain. Hence, software reliability engineering will frequently be applied to domains and

certification test within domains will increase in importance. The components may be combined into larger numbers of systems.

22. Why should you have a reliability objective? Shouldn't you strive for zero defects (faults)?

    ANSWER: Setting an objective of zero defects (faults) ignores the considerations of product delivery date and product cost. You want a balance among the characteristics of reliability, delivery date, and cost that best meets user needs.

23. Why don't you use the same definitions of terms our company uses?

    ANSWER: We use the definitions most commonly used in the literature so that you can understand and profit from the contributions of others. It would be impractical to change course or book material to accommodate all the variations of terms used by different companies and also by different organizations in the same company.

24. How do you define mean time to failure for distributed systems?

    ANSWER: The same way as you do for any system. Implicit in your question may be how you measure time. In many cases, you will use natural units rather than time. However, if you use time and have a varying average (over the current failure interval) computer utilization, so that you must measure an execution time, measuring the execution time of any processor will be satisfactory as long as that execution time maintains an essentially constant ratio with the total processing of the distributed system.

25. Shouldn't the definition of reliability take human factors into account, in that a system that is complex to use will be perceived as less reliable?

ANSWER: If the behavior of the system during execution is such that you cannot readily control it and obtain useful results from it, you can assign a failure to it, which will affect the reliability you measure. You should use this objective approach and not a subjective approach of what you perceive.

26. Don't we have some failures that are just "there" and hence have no failure intensity? For example, a name on a display is misspelled or power management software is not working properly.

ANSWER: No. A failure occurs when the unsatisfactory behavior occurs or reoccurs. If it occurs and remains for a long time, you simply have one failure with a long downtime. There *is* a failure intensity in every case: the number of failures divided by the number of natural or time units over which the product operates.

27. Is a release a separate variation?

ANSWER: It is not a separate variation in itself. But it can be a release of a separate variation.

28. We use test automation software in testing our product. Does it constitute an associated system?

ANSWER: No. Associated systems must be part of or include the delivered product. They do not include systems used to support the development of the product.

29. We are adding new features to an existing product. Does this make it a new release or a new variation?

ANSWER: It is a new release if we will ship the resulting product to all customers who request it. It is a new variation if we split our customer base and ship the product only to some

customer types who request it, shipping the existing product to the others.

30. Would a product with a capability to be operated locally or remotely represent a base product and a variation?

    ANSWER: Generally not, because the capability to operate the product either way exists in the product as you ship it.

31. Can there be several supersystems for the product and each major variation?

    ANSWER: Yes

32. Can a supersystem be concerned only with some of the operations of a base product or variation?

    ANSWER: Yes.

33. Suppose you purchase an installation disk for a word processing system and can choose to install either a basic or advanced version. Are these different variations?

    ANSWER: No, because they are delivered together as a package.

34. What is software maintainability?

    ANSWER: Note that the term maintainability has the same definition for hardware and software, the average staff hours required to resolve a failure. However, the significance is somewhat different. Resolution for hardware occurs online and affects downtime. Resolution for software occurs off line and does not affect downtime.

35. How is integrity defined?

    ANSWER: *Integrity* is the probability that a system operates without security penetration for a specified time, given a specified threat profile and rate of arrival of threats.

36. What is recoverability?

ANSWER: *Recoverability* is the average time to recover from failure, including data cleanup or reinitialization. It is the software analog to mean time to repair for hardware.

37. What is the survivability of a system?

ANSWER: *Survivability* usually refers to reliability of the system with respect to a specified set of critical functions when the system has been degraded in some way. For example, you might represent survivability of a spacecraft by the reliability of the navigation functions when the spacecraft has been subjected to radiation.

38. How is risk assessment related to software reliability?

ANSWER: Risk assessment is a study of the probability that certain undesirable events will occur. These events include software failures but can include other events as well, such as cost overruns. Thus, risk assessment can be more general than software reliability. For a detailed discussion of this area, see Li, Li, Ghose, and Smidts (2003).

39. We have a very close relationship with our software supplier. As we test their software, we send them trouble reports. They send us fixes, we integrate them, and continue test. What type of test are we performing?

ANSWER: Reliability growth test.

40. How are software quality and software reliability related?

ANSWER: Software reliability is one of the major characteristics of software quality.

41. How does software reliability engineering relate to Six Sigma?

ANSWER: Six Sigma is a goal for very high reliability. Software reliability engineering is a complete process for attaining whatever reliability you want to achieve more rapidly and more cheaply, so it is probably the best way to implement a Six Sigma goal.

42. How does software reliability engineering relate to the Capability Maturity Model?

ANSWER: The Capability Maturity Model at its higher levels requires measurement of reliability and the use of the measurements to optimize development. Software reliability engineering is the practice that can accomplish this goal.

43. How does software reliability engineering relate to Quality Function Deployment?

ANSWER: Quality Function Deployment focuses on product functions as viewed by users. Software reliability engineering, through its operational profiles, focuses not only on what the functions are but also on what their probability of use is. Thus, software reliability engineering is more advanced and sophisticated than Quality Function Deployment.

44. How can I obtain a copy of the AIAA Recommended Practice for Software Reliability (R-013-1992)?

ANSWER: You can order it over the internet as follows:

Digital Copy:

http://www.aiaa.org/store/storeproductdetail.cfm?ID=937

Paperback Copy:

http://www.aiaa.org/store/storeproductdetail.cfm?ID=194

45. Should we consider collaboratively developed software as developed or acquired software?

    ANSWER: If your organization is one of the collaborators, you should consider it as developed software and at least do reliability growth test. If not, you would consider it as acquired software and you would only do certification test.

46. How does software reliability engineering change as projects become larger?

    ANSWER: It does not change in any fundamental way, but there may be more variations and supersystems and longer operational profiles.

47. Why can't we just count faults per thousand lines of delivered executable source code instead of implementing software reliability engineering?

    ANSWER: The fault measure is a useful tool for the *developer*, and it may point up components or routines that need attention. However, software reliability engineering has additional advantages. It is *customer* oriented, as failures per thousand hours of execution relates directly to the customer's operations and costs. And software reliability can be combined with hardware component reliabilities to obtain overall system reliability.

48. If you want a customer-oriented quality measure, why don't you just use "faults found by customers" instead of failure intensity or reliability?

    ANSWER: "Faults found by customers" is only minimally useful to the customer. It does not indicate how frequently trouble will be experienced in operation, and hence it is not possible to estimate the impact or costs of failure. Most important, it only tells you what the quality is of software that has been in the

field for some time, not the quality of the new release you are just getting or about to get.

49. Since the failure intensity is proportional to the number of faults remaining, why can't we just use faults for a software quality metric instead of failure intensity?

    ANSWER: Because the faults you count are the faults *removed*, and this is very different than the faults *remaining*. You might conceivably estimate the faults remaining as a result of estimating the inherent faults. However, the accuracy of the resultant measure is much worse than that of failure intensity.

50. It seems to me that counting faults is much easier than counting failures, because data collection takes place just within the development team. When you count failures, you must collect data from testers or customers in the field. Don't you agree that it's better to take the easier approach?

    ANSWER: First, it isn't true that counting faults is easier. Failures must first be reported before the development team can work on the faults. They must be sorted and compared so that repetitions are eliminated before fault correction is initiated. The failure count is available *first*, and you can only determine the fault count after the extra processing occurs. And finally, even if counting faults were easier, the strong advantage of having a customer-oriented quality measure would outweigh any extra data collection effort.

51. My management tracks performance of system testers based on the number of faults they discover. This seems to be in conflict with the software reliability approach. But what's wrong with finding the most faults possible?

    ANSWER: Rewarding system testers for faults discovered (fault yield) will distort testing and make it inefficient because the testers will use test cases that are easiest to generate and

have the greatest promise of revealing faults. These often are "extreme value" or "boundary" input states that do not occur very frequently in operation. You should base a reward system on reducing the cost of failure in operation most rapidly (usually correlated with reducing failure intensity most rapidly, the software reliability approach).

52. Can software reliability engineering contribute to the web services area?

ANSWER: Yes. The main challenge is the intervendor coordination in defining subsystems, operational profiles, and reliability objectives.

1.6.1.3 What's been the experience with SRE?

1. How does software reliability compare in effectiveness with other approaches to quantifying quality?

ANSWER: Software reliability is the most important aspect of quality to the user, since it quantifies how well the software product will function with respect to his / her needs. Certain other measures of quality may have some relation to software reliability, but they are much less direct. For example, the number of faults remaining has some connection to reliability, but it is developer-oriented and it does not indicate impact on operation. Further, you cannot measure but only infer faults remaining, and this with rather poor accuracy. The number of faults found has NO correlation with reliability. Few faults found can indicate either reliable software or poorly tested, unreliable software. The number of problems uncovered in design or code inspection has properties similar to faults found as a quality measure. Other measures such as program complexity are even more remote from the user's concept of quality.

2. Some people say that software reliability engineering is inadequate because it does not focus sufficient attention on preventing catastrophic failures. Is this true?

   ANSWER: No, the statement represents a failure to understand software reliability engineering properly. Catastrophic failures generally represent some fraction of all failures. Hence tracking and reducing all failures will also reduce the proportion of catastrophic failures.

3. How can we use software reliability engineering to improve the design and test process?

   ANSWER: We can use software reliability engineering to improve the design and test process in several ways:

   a. We can use knowledge of the operational profile to focus design effort on the most frequently used functions.

   b. Driving test with the operational profile will activate the most-frequently used functions first, allowing us to find and remove their faults first, and thus reducing failure intensity rapidly.

   c. The failure intensity objective you establish with the customer suggests the level of design and test effort you need and what range of techniques you should employ.

   d. The failure intensity objective provides a release criterion for determining when to stop system test.

   e. Measurement of reliability differences resulting from different software engineering techniques, along with the cost and time duration associated with them, will provide the basis for selecting these techniques on future projects.

4. Can we use software reliability engineering for ultrareliable systems?

   ANSWER: Yes. Some statisticians have argued that you cannot, because the time required for testing would be impractically long. This view is based on a hardware perspective that usually does not hold for software. Ordinarily only a limited number of the software operations must be ultrareliable and they are commonly ones that do not occur frequently. With the use of appropriate software architecture such as firewalls, the operations can be isolated. Since they are usually not invoked very frequently, they can be certified to very high levels of reliability with practically achievable amounts of testing. Finally, since processing is cheap and rapidly becoming cheaper, you can speed up testing by performing it on many processors in parallel.

5. Doesn't the use of software reliability engineering in contracted development involve increased effort and cost on the part of the customer who is doing the contracting?

   ANSWER: Very little if any, and it usually reduces effort and cost because it systematizes the communication of customer needs and requirements and monitoring of the contractor's program.

6. Why do you say that software reliability engineering is more feasible for very small components if they are used as parts of multiple products? The cost of software reliability engineering relative to total project cost is the same regardless of the number of products using the component.

   ANSWER: This is correct, but it is the benefit-to-cost ratio that is important. If software reliability engineering benefits multiple products because the availability and reliability of the components are higher, and the delivery time and cost are lower, then the total benefits and hence the benefit to cost ratio are higher.

7. Doesn't it require more than software reliability engineering to produce high quality software?

   ANSWER: Of course. You need a well-defined process, change control, competent people, and the right selection of and balance among project strategies. Software reliability engineering deals with the last factor.

8. Can software reliability engineering be effective without a stable process?

   ANSWER: Yes, but it would be more effective with a stable process. The application of the operational profile will substantially improve development efficiency. However, if the process is not stable, you will not be able to choose software reliability strategies that are optimal, and hence effectiveness will not be as high as it could be.

9. I understand that you cannot improve system reliability by operating two software components in parallel, since they will experience exactly the same failures. But I've been told that you can improve availability. How is this possible?

   ANSWER: Operate the second software component in a standby mode. When the first component fails, the second will not have failed. It will not require any recovery time; it can immediately take on processing load. In theory, you will obtain a system availability of 1.0. In practice, the availability will be slightly less due to the downtime required for shifting execution to the second component.

10. Can completely duplicated (redundant) software systems ever enhance reliability?

    ANSWER: It is possible if the indirect input variables to the system differ, and only one of the values causes failures.

11. Which of the following attributes of quality can be incorporated in and measured by software reliability?

    a. Functionality (presence of features)

    b. Quality of failure messages

    c. Ease of learning

    d. User friendliness in operation

    e. Maintainability

    f. Hardware fault tolerance or recoverability

    g. Performance

    h. Extensibility

    i. Support level (speed of response to support requests)

    j. Software fault tolerance or recoverability

    k. Ease of relating user problem to system capabilities

    l. Tolerance to user faults

    ANSWER: a, b, d, f, g, j, l. We can relate all of these quality attributes to behavior of the program in execution that users may view as unsatisfactory.

12. How can you use software reliability to characterize functionality?

    ANSWER: Determine what functions your customers want or need and how often they need them. Consider the absence of a function as a failure. For example, suppose customers want a status display function of unsold seats per flight for an airline reservation system on a daily basis for making pricing decisions. You would then consider the absence of the function as a failure. Hence you have 41.7 failures / Khr in

addition to whatever other failures may exist. Although this possibility of incorporating functionality into software reliability exists, you usually don't do it. Normally, you establish a fixed set of functions for a product release and measure software reliability with respect to that set.

13. How are software safety and software reliability related?

ANSWER: Software safety is one aspect of software reliability. Reliability implies proper functioning of software, and safety is one of the requirements that must be met for proper functioning. You can categorize failures as safety-impacting or non-safety-impacting. For example, a system might have 2 safety failures per Khr. Note that an unsafe system is also unreliable.

14. Can software be unsafe without the possibility of software failures occurring?

ANSWER: Not really, if you interpret "failure" in the broad sense of "behavior that the customer will view as unsatisfactory." Certainly, an unsafe event that occurred would be interpreted as unsatisfactory. Hence, unsafe events would constitute a subset of the set of failures.

15. How does Failure Modes and Effects Analysis (FMEA) relate to software reliability engineering?

ANSWER: Software reliability engineering is a macroscopic approach that takes a global or "big picture" view of the software product involved. FMEA is a microscopic approach that looks at particular failures, how they can be caused, and how to prevent them. It is more expensive to apply than software reliability engineering due to the detailed analysis effort required. Thus, it is most practical for trying to prevent

critical failures. You can integrate the two practices as follows. Apply software reliability engineering to reduce the total failure intensity. Perform FMEA on potential severity 1 failures you can identify and implement appropriate failure prevention or fault tolerant features for them.

16. How is software reliability engineering related to cleanroom development?

ANSWER: There are many similarities between software reliability engineering and cleanroom development, but software reliability engineering is the more general practice. Cleanroom development includes hierarchical specification and design and team reviews that apply formal correctness verification. You can use software reliability engineering with cleanroom development but you can also use it with any other development methodology.

17. Does function point analysis have a role in software reliability engineering?

ANSWER: Not to any extent. Function point analysis was originally developed to support cost estimation and measures of productivity. The number of lines of code can vary widely for the same functionality, depending on a programmer's conception of the problem and his / her style. Hence, lines of code is not a good measure to use when estimating cost or measuring productivity. However, the number of faults created (and hence failure intensity) are highly correlated with the number of lines of code written, because the human error rate appears to be relatively constant per line of code. Thus, function point analysis is not needed for obtaining good predictions of failure intensity.

18. How is software reliability engineering related to stress testing?

ANSWER: Stress testing involves testing under extreme conditions such as heavy loads. Software reliability engineering includes stress testing, but also all the more common conditions as well.

19. How is software reliability engineering related to coverage test?

ANSWER: Software reliability engineering is based on use rather than coverage, so it is more truly representative of field conditions. Coverage testing is most effective at the unit level.

20. How is software reliability engineering related to risk-based test, where you test functions in order of risk?

ANSWER: Software reliability engineering is very similar, in that it tests functions in order of amount of use, adjusted by criticality. Both use and criticality are major factors in determining risk.

21. How is software reliability engineering related to statistical testing?

ANSWER: Software reliability engineering testing *is* statistical.

22. How is software reliability engineering related to user test?

ANSWER: Software reliability engineering is based on the amount of use of the different functions, where the use has been quantified. This is usually more precisely representative of actual use than turning several users loose with the program, because they may not represent a sufficiently large or representative sample of the user population.

23. How is software reliability engineering related to scenario testing?

ANSWER: Scenario testing simply involves testing typical sequences of functions. It is not as representative as software reliability

engineering test, because it does not take account of the relative amounts of use of the functions.

24. How does software reliability engineering relate to ISO standards?

ANSWER: ISO standards are designed to make you document the software development process you are using and to check compliance with the process you document. This process does not necessarily have to be good or competitive. On the other hand, incorporating the software reliability engineering process in your software development process yields an extremely advanced, effective, competitive process.

25. Can software reliability engineering be used with extreme programming?

ANSWER: Yes. Since extreme programming is a methodology of agile software development, we will look at the larger question of how software reliability engineering can be used with agile software development (next question).

26. Can software reliability engineering be used with agile software development?

ANSWER: Yes. As you may know, agile software development is software development that emphasizes rapid responsiveness to changing requirements. In reality, we do not have two kinds of software development, agile or nonagile, but rather a continuum of agility, although there is some tendency to bipolarity.

Agility becomes increasingly important when customer needs and/or technological opportunities change rapidly so that more exploration during development is required. Agility implies faster and cheaper development, with the possibility of trading off reliability to get it. This requires feature-oriented development, faster and shallower planning, and

greater and faster interaction among customers and developers.

Software reliability engineering very precisely and efficiently guides you in making the time-cost-reliability tradeoff, with the result that you make a tradeoff that is much closer to customer needs. The tradeoff can be quickly modified if conditions change. The operational profile inherent in software reliability engineering supports operational development, which is a quantitatively guided (and hence greatly superior) type of feature-oriented development. Operational profiles are rapidly developed and modified with the use of spreadsheets, making software reliability engineering a very agile supporting practice. Test case creation and execution can be driven with the spreadsheets, resulting in very agile testing.

27. How does software reliability engineering differ from other industry processes?

ANSWER: It does *not* differ. You do not choose software reliability engineering instead of other good processes and practices. You add it to and integrate it with these processes and practices, and use software reliability engineering to guide them.

28. Is there a minimum CMM level or are there other minimum process requirements necessary before you can implement software reliability engineering?

ANSWER: You need some order in your process and you need to do unit test, so perhaps you need to be at level 2. However, you can implement software reliability engineering simultaneously with these requirements.

29. Can software reliability engineering be applied to object-oriented development?

ANSWER: Yes

30. Why should we set a failure intensity objective? Why don't we just develop an operational profile and use it to allocate our development resources and make our testing realistic?

    ANSWER: The operational profile will help you greatly in allocating development resources and conducting realistic tests, but it can't guide you in determining how long you should test or how much effort you should put into reliability improvement strategies. You need an absolute measure like a failure intensity objective to help you there.

31. How can you say that higher software reliability costs more? Quality theorists say that higher quality costs less.

    ANSWER: The quality theorists are referring to the reduction of defects in manufactured products through higher quality of the manufacturing process. This reduces costs because of the reduction in rework needed to give the same delivered quality to customers. The analog in software development is that prevention of faults costs less than removal to get the same level of reliability. But a higher level of reliability does cost more.

1.6.2 The software reliability engineering process

1. When do you involve users in software reliability engineering?

   ANSWER: You involve users in setting product and supersystem failure intensity objectives and in developing operational profiles. If users specify acceptance tests, they will be involved in them.

2. Why do we need to collect data concerning failures?

   ANSWER: So you know what reliability your customers or users are experiencing.

2. Can software reliability engineering help you in determining the requirements for a product?

   ANSWER: Yes. Record and analyze the field operational profile for a similar product or a previous release of the same product, paying particular attention to variations among different users. The way the product is actually used may not be the same as you expected. Differences in application among users may suggest ways in which they could better use it. Better understanding of use will lead to better requirements.

3. In our application, my management tells me that our objective is perfection with respect to certain functions that are critical to success of our overall mission (success of all of these functions is necessary for mission success; none can fail). How can we apply software reliability measures?

   ANSWER: Expecting perfection is unrealistic, although admitting that your goal is less than perfection may be difficult from a public relations standpoint. What you really want to do is to determine the degree of "near perfection" needed.

   You will be in good company, because developers of "life critical" software such as that used in medical systems, aircraft and air traffic control, and nuclear power plants deal with such questions.

   Setting an appropriate objective for the critical functions generally depends on the impact of failure, usually in terms of human life or cost. Although it may be difficult to set precise objectives, it is usually possible to come up with bounds or approximate objectives. An approach that is frequently useful is to look at past experience or similar situations to estimate what is acceptable. For example, in the United States, people are apparently willing to accept the current risk level with respect to automobile use of about

one fatal accident per 10^6 hr of operation, because this risk does not influence people to stop driving.

An annual failure cost level that represents an appreciable fraction of a company's annual profits would probably be considered unacceptable, as would a failure risk level that would result in serious public demand to stop the operations supported by the system you are considering.

4. Do failure measures at different phases of test (for example, unit test, subsystem test, and system test) have equal significance?

ANSWER: Measures taken later in test generally will give a better indication of what to expect in operation. This is because more data is available, and the data that is available is usually based on more of the system being present and on a test operational profile that is closer to the true operational profile.

However, this does not mean that data taken early in development will not give useful approximations of what to expect in operation. You must adjust early data to account for only part of the system being present and for test operational profiles that depart from the actual operational profiles.

5. Are attaining reliability growth and evaluating the reliability that exists conflicting objectives?

ANSWER: Probably not, but if they are, then only to a limited extent. Evaluating existing reliability requires that you represent field conditions accurately when you test; i.e., you select your tests in accordance with the operational profile.

Attaining optimal reliability growth implies that you select your tests so that you increase reliability per unit test time as rapidly as possible. To a certain degree, selecting tests in accordance with the operational profile gives

the most rapid reliability growth, because you will test frequently executed code early and find and remove any faults that lurk within it. However, you may also repeatedly exercise some frequently executed code before less frequently executed code is exercised for the first time.

If the software operates in a limited environment (a limited variation of indirect input variables, as defined in Chapter 4), the repeated execution may result in removal of almost all the faults in that code. Then continued execution of the code doesn't result in much further reliability growth. Thus, it is possible that an approach other than pure operational-profile-driven selection could improve test efficiency later in test. However, no specific proven approach exists at this time.

6. Can you apply software reliability engineering to systems that are designed and tested top down?

   ANSWER: Yes. When applying software reliability engineering to system test, you will probably have a shorter test period because there will be fewer faults to remove. However, it seems likely that there will always be a substantial system test period before the release of any software product. It will always be necessary to run the code through a number of functional tests.

7. What are the most common mistakes people make in applying software reliability engineering?

   ANSWER: Two of the most common are:

   a. In constructing the operational profile, they miss some operations with significant probabilities of occurrence because they neglect to consider special or transient environments (for example, a data base in the process of being populated).

b. They do not test in accordance with the operational profile. Reliability estimates obtained then do not reflect actual expected use.

8. Do any special changes have to be made to apply software reliability engineering to an object-oriented program?

ANSWER: No

9. What changes in the "traditional" software development process are necessary to implement software reliability engineering?

ANSWER: The changes are not substantial and they carry many additional benefits beyond software reliability engineering itself:

a. You need to determine one or more operational profiles for the product.

b. System testing must be consistent with the operational profile.

Only the first change involves any appreciable effort. The establishment of an operational profile will let you focus development on those functions that are most frequently used, with a strong net gain in productivity. The change in system test represents a change in order of test rather than magnitude of effort. Software reliability engineering requires that you record test output and examine it for failures, but this is not a new requirement above good conventional system testing practice.

Incidentally, software reliability engineering does not affect the work of system architects and developers, except to make it more efficient, unless perhaps there is a desire to build failure recording and reporting features into the delivered product.

10. Some of our project people are uncomfortable with the idea of suggesting that customers set failure intensity objectives. How do you deal with that?

    ANSWER: Point out that it is to the long term benefit of both they and their customers that they understand customer requirements in specific quantitative terms. Of course, this assumes that both understand the tradeoffs involved among reliability, time of delivery of new features, and cost. Better understanding means that the customer will be more likely to be satisfied and that the development organization will be more likely to predominate vis-à-vis its competitors. Although there may be some initial resistance to letting customers set failure intensity objectives, this should disappear with greater understanding.

11. Can we apply software reliability engineering to unit testing of a product?

    ANSWER: In theory, yes. But it is generally not practical to do so unless you expect to use the unit in multiple products, because of the effort and cost required to develop an operational profile for the unit. Using software reliability engineering to certify components expected to be used in multiple products is highly desirable, however.

12. We do adaptive testing; that is, we wait and specify some of our test cases in the test phase after we see initial results of some of the tests. Will this influence the way we will apply software reliability engineering?

    ANSWER: You will apply the same basic principles and practices. However, since you will be delaying preparation of some of the test cases until the test phase, you will need additional resources at this time.

13. Can software reliability engineering be applied to third party test (test by a person or organization external to the company developing the product)?

    ANSWER: YES. In fact, the value added by software reliability engineering is even greater in this case. It provides an organized, systematic, measurable approach to testing. Such an approach is particularly important for third party testing, where good management and communication is necessary to overcome the extra risk involved.

14. How do you apply software reliability engineering to a program that you integrate and test in major stages?

    ANSWER: Apply it in the conventional way, but do not process failure data from testing performed before all stages of the program are present. It is actually possible to handle this early data if you wish (Section 6.4.1) but it probably is not worth the extra trouble, because failure intensity estimates are of low accuracy.

15. Is structural testing included in software reliability engineering testing?

    ANSWER: No, if by structural testing you mean systematic testing of program branches and paths without regard to probability of use. Software reliability engineering testing is usage-based and requirement-based.

16. How does software reliability engineering handle intervendor testing?

    ANSWER: It tests supersystems that incorporate the products of different vendors (see Chapters 1,3, and 5).

17. Does software reliability engineering include performance test?

    ANSWER: There is no requirement that it do so. However, since software reliability engineering test and performance test both require realistic

operational profiles, it can save money to execute them together.

18. Does software reliability engineering include integration test?

ANSWER: This depends on what you include in integration test. Software reliability engineering does not include tests of interfaces nor does it include test that occurs as an operation is put together. Software reliability engineering begins as soon as an operation can be executed as a whole.

19. Does software reliability engineering include subsystem test?

ANSWER: Not as an activity in the process for the product being tested. However, a subsystem of a product can itself be a product. In that case, software reliability engineering can be applied just as it is to any other product.

20. What is the best way to compare the reliability growth performance of several completed projects; and, what is your recommended method for using this historical data to improve forecasting of reliability growth for "in progress" projects?

ANSWER: The concept of "compare" among projects is questionable because they differ greatly in the balance of reliability vs delivery time vs cost that customers need. If this balance is the same for different releases of a single project, you might possibly compare them. Using historical data to improve estimation is dubious, but you might use it to improve prediction. See Chapter 6.

21. How do we get started applying software reliability engineering?

ANSWER: For information on deploying software reliability engineering, see Chapter 7.

22. I notice that when you talk about applying software reliability engineering to legacy systems you always talk about doing it "for the

next release." Can't you just apply software reliability engineering to the product without waiting for the next release?

ANSWER: Not efficiently, because software reliability engineering is a *process* that you integrate with your software development process. If you aren't doing anything, there is no process to engineer with your failure intensity objectives. This does not mean that the next release must have new operations or features. It could be a release that simply has the goal of improving reliability through extensive testing and debugging. Software reliability engineering will enable you to do this more efficiently, more rapidly, and less expensively by focusing you on the operations that are used the most and are most critical and by measuring the improvement so that you don't continue it past the point of diminishing returns.

23. At present, only our test team is willing to implement software reliability engineering and we cannot persuade other organizations to collaborate. Can we just deploy parts of software reliability engineering?

ANSWER: Yes, as long as you deploy parts where you can take the initiative yourself in obtaining the information you need

24. How do you design for reliability?

ANSWER: The term "design" is often confused with the concept "establish a good development process for." We will interpret the more restrictive meaning, which relates primarily to fault tolerant features of the software. You must determine a fault tolerance failure prevention goal based on a determination of the failure intensity objective and the choosing of software reliability strategies. Using the operational profile, determine for which operations you must install fault tolerant

features and the percent failure preventions required, such that you minimize the design and operational costs of the fault tolerant features. Design to these goals.

25. Does software reliability engineering require an independent test group?

    ANSWER: No, but it encourages it. The need for an operational profile encourages communication with users. Since developers are heavily occupied with development itself, an independent test group can usually better fill this role.

26. Can software reliability engineering be effective when the development process is not stable?

    ANSWER: Yes, if you apply it properly, because it will help stabilize the process.

27. How should you handle "reliability improvement" releases, i.e., releases that have no new operations but represent solely an attempt to improve reliability through increased system test?

    ANSWER: Consider a reliability improvement release as simply a "reopening" of a release to additional system test.

28. Can fault tree analysis be used to predict software reliability?

    ANSWER: Not in any practical sense. Fault tree analysis is a microscopic approach to software reliability, one that is focused on individual failures rather than macroscopic behavior.

29. Does the practice of software reliability engineering vary from country to country?

    ANSWER: No.

30. Does the practice of software reliability engineering vary from industry to industry?

ANSWER: Not really. The principal variation is from product to product, but even here, the basic principles are the same. You merely make small adjustments and adaptations due to product variations. This does *not* mean that operational profiles, major quality characteristics, or software reliability strategies will be the same from product to product.

31. How can software reliability be used to characterize security of a system?

ANSWER: Security of a system is dependent on several software-based functions, which include access control (authentication of identity) to both operation and to code, input checking, and encryption / decryption. The reliabilities of these functions are measures that indicate the level of security that we have.

32. How can we relate system reliability requirements to reliability required for a software component?

ANSWER: This is discussed in Chapter 3, where we learn how to find the software failure intensity objective needed for the software we will develop, given the expected failure intensities of hardware components and acquired software components.

33. How are the software reliability engineering and software development processes affected when you have staged builds?

ANSWER: Design and implementation in the software development process produce staged builds which are released to test. Each of these builds will be associated with a set of test cases that have been prepared for it, the test cases for the operations that are valid for that build. The test cases will have been prepared during the test preparation activity of the software reliability engineering process. They will be executed during the test execution

activity of the software reliability engineering process, which corresponds to the test phase of the software development process.

34. How does software reliability engineering address interoperability requirements?

    ANSWER: By defining and testing supersystems.

35. How can we keep the amount of supersystem test we perform within reasonable limits?

    ANSWER: Establish thoroughly and carefully defined interfaces for independent systems you expect to interact with and provide strong motivations for developers of these systems to conform to these interfaces. Supersystem testing will theoretically not be necessary for such systems.

    In practice, there is some risk in relying on the thoroughness of the interface definitions and the conformance of other developers to them. Hence, you will generally use this approach to reduce rather than eliminate supersystem test for the systems identified.

36. When we apply software reliability engineering, which types of test can we include?

    ANSWER: Many types of test have been separately named. You can apply software reliability engineering to all types of test that, taken *together* over a time interval (let's say, over a day), follow the operational profile. Thus, it can include feature test, regression test, configuration test, load test, stress test, stability test, bang-bang test, operational scenario test, performance test, and security test. It usually excludes installation test.

37. What is failure modes and effects analysis (FMEA)? When might I want to use it for software?

ANSWER: Failure modes and effects analysis looks at component failures and determines what kind of system failures can result. FMEA based on specific failures is probably not practical for software; it you knew what specific failure would occur, you would fix it during the coding phase. However, conducting FMEA at the module level can help you identify which modules are most critical such that systems operations work properly. This can help you focus design review, code review, and unit test efforts. Note that a given module can have a range of system failure severities that result from its failures; hence you will be identifying criticality in an average sense.

Since a FMEA can involve considerable effort and cost, you will probably want to limit employing it to highly used and/or most critical operations. The operational profile can be a valuable aid in helping you identify these.

38. What is the difference between a component and one of the independent systems of a supersystem?

ANSWER: A component is part of the product you are selling; you take legal responsibility for the correct operation of the product including the component. You do not sell or have legal responsibility for the operation of the independent systems of a supersystem, even though they work with your product. However, since customers expect that your product will operate reliably as part of the supersystem, you expend effort testing the supersystem.

39. Use of failure-oriented rather than fault-oriented quality measurements is fine in theory, but it depends on taking field measurements. How can we get them, because we can't put the reporting burden on customers?

ANSWER: It might be a problem if we required customers to record failures manually. But we can automate the process, with most failures of higher severities being recognized by

appropriate verification or auditing software. Reporting to a central data collection point can also be automatic. The principal problems that can occur are excessive overhead from the recording software and the inability to recognize all failures. The first problem is diminishing in importance as computer throughput increases. One can compensate for the second problem if the proportion of failures missed remains approximately constant with time.

1.6.3 Defining the product

1. Can you beneficially apply software reliability engineering to "one shot" systems that you rapidly construct in real time from off-the-shelf components and use for just a limited period? Illustrations include user-programmed systems and agents and applets on the Internet.

   ANSWER: Yes, although clearly not in the same sense you would apply it for commercial software you expect to have substantial use. A useful analysis would be to determine the reliability level to which you would have to build the components such that systems of acceptable reliability could be constructed.

2. Can you apply software reliability engineering to systems with a low production volume?

   ANSWER: Yes. The only difference is that the product budget will probably be lower. Hence, the resources available for test will be less. As a result, you may feel more pressure to keep the number of systems tested, the number of operations, the number of test cases, and the amount of test time low.

3. Can a product be an object or an off-the-shelf component?

   ANSWER: Yes. A product is a system or a set of associated systems that an organization sells and delivers. It can be very large (millions of lines of code for the software part) or quite

small, as long as it is produced and sold separately.

4. Does reliability have any particular psychological effects on users of a system that need to be considered?

   ANSWER: When users first begin to employ a new system, it is important that the reliability of frequently used functions be high. If they are not, the system will rapidly gain a bad reputation that may later be difficult to overcome. Users may avoid using parts or all of the system or attempt to work around it. This situation is another powerful argument for determining the expected use for a system and using it to guide testing, with the most frequently used functions tested first.

5. Flight software used on space missions is often "one of a kind." Does this present a problem in software reliability engineering application in that previous experience may not be applicable to the next mission?

   ANSWER: No. Estimates of failure intensity during test will be based on failure data, which will depend on this and not previous missions.

6. Can reuse of software be helpful from a reliability as well as a cost standpoint?

   ANSWER: Yes. Reused software components tend to have a higher reliability, resulting in higher system reliability. But you must verify the reliability of a reused component for the system in which you will employ it and for the use and environment of that system.

7. What differences must you consider in applying software reliability engineering to packaged software (i.e., software that is sold to a large volume of customers)?

   ANSWER: Initial sales of packaged software probably depend more on getting a lot of *useful* functionality to market ahead of competitors.

Hence, rapid testing of the most used features to some minimum standard of reliability is very important. Meeting some minimum standard is essential, because if the product fails too often, it will get a bad reputation from which it may never recover. But you should not test any longer than you have to, because otherwise a competitor may deliver a product incorporating the most used features before you do. Achieving good reliability for the lesser-used features can await later releases. Thus, operational development (discussed in Chapter 2) is an excellent approach.

When you set the ultimate failure intensity objective to be achieved, the tradeoff with cost will be particularly important due to the severe competition on price that is common.

With the large numbers of users typical for packaged software, you will need to employ sampling to get operational profile data and data on reliability needs. You may have to provide some inducement to users to provide the data. Some software suppliers have included use recording with beta test releases provided free to early users. These releases are often distributed over the Internet. Note that you must take care in interpreting data from a special group of users such as this; they may not be representative of the total eventual user population.

8. Our product must operate with a wide variety of interfacing hardware and software. How do we organize testing in this situation?

ANSWER: Each of the configurations of hardware and software that your product must work with forms a supersystem when combined with your product. You need to test each of these supersystems.

You can see that it is quite easy for the number of supersystems to get out of hand. You may wish to develop a *configuration profile* [Juhlin (1992)], which is a list of

configurations and their probabilities of occurrence. These probabilities are usually determined as fractions of users associated with a particular configuration. This data is generally a reasonable approximation to the fraction of use and is easier to collect than the fraction of use. You can use the configuration profile to allocate your testing effort, even to the extent of not testing rarely occurring configurations.

Another strategy is to define a base configuration plus a set of delta configurations. Each delta configuration represents only the changes that make up a particular variation from the base configuration. Many of the deltas will affect only a limited set of operations. You can often save a great deal of testing time by testing the base configuration for all operations while testing the delta configurations for only those operations associated with them.

9. We produce a notebook computer, writing some basic firmware and software that lets a standard commercial operating system run on it. The product must work with various printers. How do we test the computer with them?

   ANSWER: Plan a set of supersystem tests, each test comprising the computer and one of the printers.

10. One of the components of our system is no longer supported. Should we perform certification test on it?

    ANSWER: Yes, if you have not done so already. You need to know its reliability and how that will affect your system reliability. If the effect is unacceptable, you will have to find an alternative component or some way of increasing the reliability of the existing one.

11. Can we apply software reliability engineering to legacy systems? When?

ANSWER: Yes, beginning at the start of any release cycle.

12. What types of systems are typically deferred to beta test because they can't be economically tested in system test?

    ANSWER: Supersystems, because it is often very expensive to reproduce or even simulate the systems that interact with your product. This is particularly true for networks of systems that extend over a substantial geographical area.

13. When we test interfaces, should we use certification or reliability growth test?

    ANSWER: When you test an interface, you usually test it in two ways: in each of the interfacing systems that it spans, checking the variables transmitted, and as the system that includes both of the interfacing systems, when you observe overall behavior. The type of test is the same as the type of test used for the system in question.

14. How does software reliability engineering apply in an organization that uses a layered architecture for all its products: user interface, application, and database?

    ANSWER: The activities involved should essentially be the same. Some of the layers, such as the database and the user interface, will probably be products in their own right. Suppliers will combine them with applications to yield "final" products and deliver these products to "final" users. Note that products such as the database and the user interface may function with many other systems. Hence, you may need to consider a number of external systems as initiators of operations when developing operational profiles. Also, you can expect a need to test a number of supersystems.

15. We integrate a variety of operating systems and COTS software to work together. Our customer specifies all these software products; we have no role in choosing them. Our job is to intercept failures in the products or the interfaces and counteract them. We wrap or constrain the functions of software we can't control. How can we apply software reliability engineering?

    ANSWER: You are actually doing more than just integrating components. Although you are not removing faults, you are taking actions to counteract them! Thus, you are doing more than certifying the overall system. You are in effect building fault tolerance into the overall product on an ad hoc basis.

    The actions you are taking will result in reliability growth. Hence, you should choose a failure intensity objective and track failure intensity against it. You will also need to develop one or more operational profiles to characterize use of the overall system. Consequently, you will be applying software reliability engineering in essentially the conventional way. The only differences are that you have no role in choosing the components of the system and that you are constrained in the actions you can take to resolve failures.

16. Can software reliability engineering help in designing interoperable systems?

    ANSWER: Yes. By defining all the supersystems in an orderly fashion, you are much less likely to forget one. Also, the operational profile associated with the base product or variation that supports a supersystem will help you focus your resources on the most used and critical operations.

17. Can software reliability engineering be applied to dynamically composable systems (systems whose components are selected and linked in real time)?

ANSWER: Yes. You should list the systems that are possible, their failure intensity objectives, and their components. Determine the failure intensity objective required for each component in order to meet the failure intensity objectives of all the systems it can be a part of. Then apply software reliability engineering to each component.

18. How do you apply software reliability engineering when the system you are developing can never be executed in the real world: for example, a ballistic missile defense system?

ANSWER: Your question is not really raising an software reliability engineering issue but a testing issue. The testing issue is typically resolved by executing the system with a high quality simulator. You then apply software reliability engineering in the simulated environment just as you would in the real world environment. The validity of what software reliability engineering tells you will primarily depend on the quality of the simulation.

19. Can software reliability engineering be used with distributed systems?

ANSWER: Yes.

20. Can you use software reliability engineering when you test web sites?

ANSWER: It depends on what you mean by testing websites. Software reliability engineering can be used in testing a program that executes on a website, just as it can be used in testing any program. However, software reliability engineering does not offer any particular help in testing performance characteristics such as speed of download and capacity in terms of number of users. Similarly, it does not provide any special help in testing out all the hyperlinks.

21. How do you apply software reliability engineering to user-programmable software?

    ANSWER: Your product is only the part of the software that you deliver to users; you can only develop operational profiles and be responsible for the reliability of this part. The users may want to apply software reliability engineering to the various products they create by programming your product. In that case, each product will consist of a component acquired from you and a component the user develops.

22. Can software reliability engineering apply to components of software systems?

    ANSWER: Yes. The only problems arise in applying software reliability engineering to a very large number of small components: costs increase and tracking of failure intensity during test may be less accurate because you may have only a small sample of failures.

23. How is the application of software reliability engineering the same or different when applied to continuous mode operation systems versus demand mode operation systems?

    ANSWER: There is no essential difference.

24. Can software reliability engineering be used with embedded systems?

    ANSWER: Yes.

25. Why don't we fully apply software reliability engineering to every component or subsystem we have?

    ANSWER: It may not be cost effective to do this. Each such component or subsystem requires one or more operational profiles, and the cost of gathering the necessary data may not be justified by the benefits. If the component or subsystem is used in several products, however,

it may be cost effective to fully apply software reliability engineering.

Note that we may profitably apply software reliability engineering to small modules in another way, however. Develop the operational profiles for a large product, along with its operation-module matrices. Use this information to determine module usage tables. You can then use module usage tables to allocate development, code, and test resources among modules so as to cut schedules and costs (Section 2.4.3).

26. Our product must work with three different categories of systems, and each category has two possibilities. Is there a way to systematize the determination of supersystems?

ANSWER: Yes. You need to enumerate all possible combinations. Construct a table, with a column assigned to represent each category. The possibilities within each category are entered in the cells of that column. The resulting rows represent the possible supersystems. Some of the combinations may not actually occur; eliminate these rows.

For example, suppose we have a printer that must work with the Windows XP and Macintosh operating systems, with servers S1 and S2, and networks N1 and N2. We form the supersystem table in Table 1.5.

Assume that use probabilities for the other systems the printer interfaces with are Windows XP, 0.9; Macintosh, 0.1; S1, 0.8; S2, 0.2; N1, 0.7; N2, 0.3. You can now calculate the relative use of the supersystems as noted in Table 1.6.

You can use this information to allocate test time among the supersystems (you would be implicitly assuming that they present equal risks).

Alternatively, you may decide to combine test in some way for some of the low use supersystems that don't differ too greatly. You may even eliminate test for very low use supersystems.

Table 1.5 Supersystem table

Operating system	Server	Network
Windows XP	S1	N1
Windows XP	S1	N2
Windows XP	S2	N1
Windows XP	S2	N2
Macintosh	S1	N1
Macintosh	S1	N2
Macintosh	S2	N1
Macintosh	S2	N2

Table 1.6 Use of supersystems

Oper. sys.	Server	Network	Probability
Windows XP	S1	N1	0.504
Windows XP	S1	N2	0.216
Windows XP	S2	N1	0.126
Macintosh	S1	N1	0.056
Windows XP	S2	N2	0.054
Macintosh	S1	N2	0.029
Macintosh	S2	N1	0.014
Macintosh	S2	N2	0.006

27. What characteristics of a website affect users' perceptions of it?

ANSWER: The most important characteristics affecting user satisfaction are availability, wait time, download time, reliability, content quality, and user friendliness. Availability and reliability are two of the major quality characteristics that apply to all software-based products. Wait time depends primarily on traffic level and server capacity. Download time is primarily affected by the bandwidth of the user's link; however, the website can have some

influence by minimizing file sizes. Content quality and user friendliness are important website design requirements.

28. Is there any difference between reliability of a website program and reliability of other software-based products?

    ANSWER: No. For example, an e-commerce website would have operations very similar to those of a retail "bricks and mortar " store: customer registration, merchandise search, transaction, credit verification, inventory control, etc. Hence, all the technology and practices of software reliability engineering apply.

29. If an acquired software component is not assurable because we don't have time to test it, or we don't believe that the supplier will take any action if we reject it, even though its reliability is important, do we just ignore the possible problem?

    ANSWER: No, we are just limited in our options. The solution that is probably most feasible if trouble occurs is to find a workaround for the function provided by the component. A second possibility is to find an alternative supplier.

30. Is assurable acquired software still acquired software if you resolve one of its failures by removing the fault that is causing it?

    ANSWER: Yes. Whether a system is acquired or developed is a question of the degree of effort you devote to it. It remains acquired if you resolve a small number of failures and do not assume responsibility for its performance. It becomes developed when you resolve the majority of its failures and assume responsibility for its performance.

31. Do we consider software that has been openly and collaboratively developed (for example, LINUX or other open software) as acquired or developed?

ANSWER: Generally you should consider it as acquired, unless you assume the majority of the responsibility for its development.

32. If a particular release of a product involves a new release of assurable acquired software, should that new release be tested as assurable acquired software also?

ANSWER: Yes, if the new release is sufficiently different from the first release of the assurable acquired software such that its reliability is uncertain.

33. Can't you use the same test cases for assurable acquired software and the base product or variation?

ANSWER: Not necessarily. You target the test cases for an assurable acquired software to test that system alone, with the rest of the base product or variation not necessarily present. You can implement them so that they test the assurable acquired software through a skeleton representation of the base product or variation, using the operational profile for the same base product or variation rather than the operational profile for the assurable acquired software. However, the latter approach is not always the case, hence you must assume that the system has its own test cases.

34. Should larger acquired software have more test cases?

ANSWER: In general, yes, but the proportion of time during which you execute them is the most important factor.

35. Should we test one of the systems of the supersystem that our product must work with as acquired software?

ANSWER: No, it is not acquired software because you do not incorporate it as part of your product and take responsibility for its correct functioning. Further, it is not assurable

because you have no power to correct any deficiencies that it might have.