Software Reliability and Quality Management

Software Reliability Issues

Specific Instructional Objectives

At the end of this lesson the student would be able to:

- Differentiate between a repeatable software development organization and a non-repeatable software development organization.
- What is the relationship between the number of latent errors in a software system and its reliability?
- Identify the main reasons for why software reliability is difficult to measure.
- Explain how the characteristics of hardware reliability and software reliability differ.
- Identify the reliability metrics which can be used to quantify the reliability of software products.
- Identify the different types of failures of software products.
- Explain the reliability growth models of a software product.

Repeatable vs. non-repeatable software development organization

A repeatable software development organization is one in which the software development process is person-independent. In a non-repeatable software development organization, a software development project becomes successful primarily due to the initiative, effort, brilliance, or enthusiasm displayed by certain individuals. Thus, in a non-repeatable software development organization, the chances of successful completion of a software project is to a great extent depends on the team members.

Software reliability

Reliability of a software product essentially denotes its trustworthiness or dependability. Alternatively, reliability of a software product can also be defined as the probability of the product working “correctly” over a given period of time.

It is obvious that a software product having a large number of defects is unreliable. It is also clear that the reliability of a system improves, if the number of defects in it is reduced. However, there is no simple relationship between the observed system reliability and the number of latent defects in the system. For example, removing errors from parts of a software which are rarely executed makes little difference to the perceived reliability of the product. It has been experimentally observed by analyzing the behavior of a large number of programs that 90% of the execution time of a typical program is spent in executing only 10% of the instructions in the program. These most used 10% instructions are often called the core of the program. The rest 90% of the program statements are called non-core and are executed only for 10% of the total execution time. It therefore may not be very surprising to note that removing

60% product defects from the least used parts of a system would typically lead to only 3% improvement to the product reliability. It is clear that the quantity by which the overall reliability of a program improves due to the correction of a single error depends on how frequently is the corresponding instruction executed.

Thus, reliability of a product depends not only on the number of latent errors but also on the exact location of the errors. Apart from this, reliability also depends upon how the product is used, i.e. on its execution profile. If it is selected input data to the system such that only the "correctly" implemented functions are executed, none of the errors will be exposed and the perceived reliability of the product will be high. On the other hand, if the input data is selected such that only those functions which contain errors are invoked, the perceived reliability of the system will be very low.

Reasons for software reliability being difficult to measure

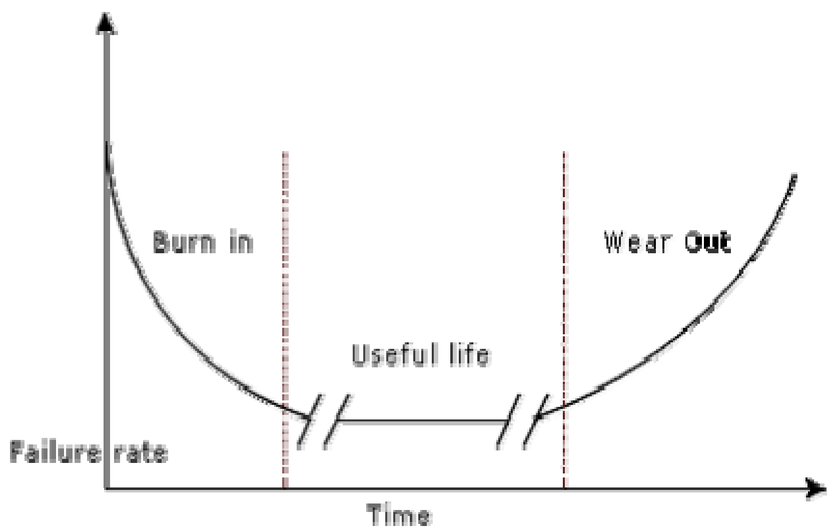
The reasons why software reliability is difficult to measure can be summarized as follows:

- The reliability improvement due to fixing a single bug depends on where the bug is located in the code.
- The perceived reliability of a software product is highly observer-dependent.
- The reliability of a product keeps changing as errors are detected and fixed.

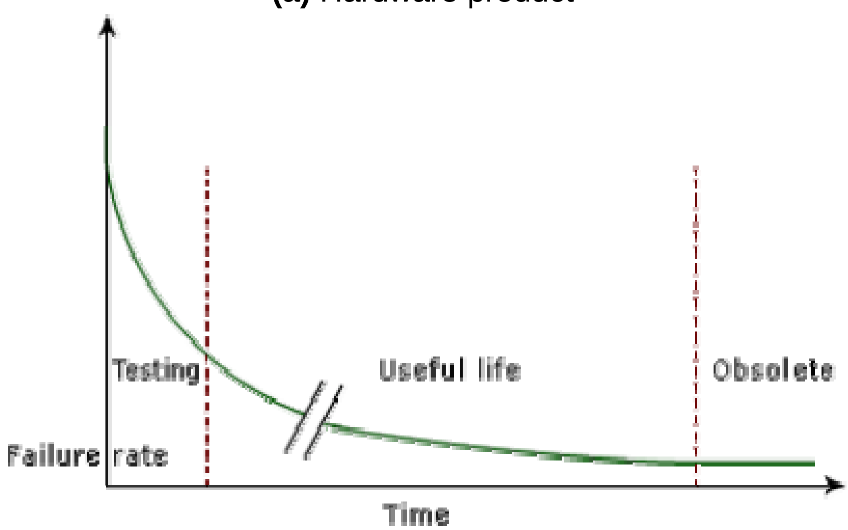
Hardware reliability vs. software reliability differ

Reliability behavior for hardware and software are very different. For example, hardware failures are inherently different from software failures. Most hardware failures are due to component wear and tear. A logic gate may be stuck at 1 or 0, or a resistor might short circuit. To fix hardware faults, one has to either replace or repair the failed part. On the other hand, a software product would continue to fail until the error is tracked down and either the design or the code is changed. For this reason, when a hardware is repaired its reliability is maintained at the level that existed before the failure occurred; whereas when a software failure is repaired, the reliability may either increase or decrease (reliability may decrease if a bug introduces new errors). To put this fact in a different perspective, hardware reliability study is concerned with stability (for example, inter-failure times remain constant). On the other hand, software reliability study aims at reliability growth (i.e. inter-failure times increase).

The change of failure rate over the product lifetime for a typical hardware and a software product are sketched in fig. 13.1. For hardware products, it can be observed that failure rate is high initially but decreases as the faulty components are identified and removed. The system then enters its useful life. After some time (called product life time) the components wear out, and the failure rate increases. This gives the plot of hardware reliability over time its characteristics "bath tub" shape. On the other hand, for software the failure rate is at it's highest during integration and test. As the system is tested, more and more errors are identified and removed resulting in reduced failure rate. This error removal continues at a slower pace during the useful life of the product. As the software becomes obsolete no error corrections occurs and the failure rate remains unchanged.



(a) Hardware product



(b) Software product

Fig. 13.1: Change in failure rate of a product

Reliability metrics

The reliability requirements for different categories of software products may be different. For this reason, it is necessary that the level of reliability required for a software product should be specified in the SRS (software requirements specification) document. In order to be able to do this, some metrics are needed to quantitatively express the reliability of a software product. A good reliability measure should be observer-dependent, so that different people can agree on the degree of reliability a system has. For example, there are precise techniques for measuring performance, which would result in obtaining the same performance value irrespective of who is carrying out the performance measurement. However, in practice, it is very difficult to formulate a precise reliability measurement technique. The next base case is to have measures that correlate with reliability. There are six reliability metrics which can be used to quantify the reliability of software products.

- **Rate of occurrence of failure (ROCOF).** ROCOF measures the frequency of occurrence of unexpected behavior (i.e. failures). ROCOF measure of a software product can be obtained by observing the behavior of a software product in operation over a specified time interval and then recording the total number of failures occurring during the interval.
- **Mean Time To Failure (MTTF).** MTTF is the average time between two successive failures, observed over a large number of failures. To measure MTTF, we can record the failure data for n failures. Let the failures occur at the time instants t_1, t_2, …, t_n. Then, MTTF can be calculated as $\sum_{i=1}^{n} \frac{t_{i+1-}t_i}{(n-1)}$. It is important to note that only run time is considered in the time measurements, i.e. the time for which the system is down to fix the error, the boot time, etc are not taken into account in the time measurements and the clock is stopped at these times.
- **Mean Time To Repair (MTTR).** Once failure occurs, some time is required to fix the error. MTTR measures the average time it takes to track the errors causing the failure and to fix them.
- **Mean Time Between Failure (MTBR).** MTTF and MTTR can be combined to get the MTBR metric: MTBF = MTTF + MTTR. Thus, MTBF of 300 hours indicates that once a failure occurs, the next failure is expected after 300 hours. In this case, time measurements are real time and not the execution time as in MTTF.
- **Probability of Failure on Demand (POFOD).** Unlike the other metrics discussed, this metric does not explicitly involve time measurements. POFOD measures the likelihood of the system failing when a service request is made. For example, a POFOD of 0.001 would mean that 1 out of every 1000 service requests would result in a failure.

- **Availability.** Availability of a system is a measure of how likely shall the system be available for use over a given period of time. This metric not only considers the number of failures occurring during a time interval, but also takes into account the repair time (down time) of a system when a failure occurs. This metric is important for systems such as telecommunication systems, and operating systems, which are supposed to be never down and where repair and restart time are significant and loss of service during that time is important.

Classification of software failures

A possible classification of failures of software products into five different types is as follows:

- **Transient.** Transient failures occur only for certain input values while invoking a function of the system.
- **Permanent.** Permanent failures occur for all input values while invoking a function of the system.
- **Recoverable.** When recoverable failures occur, the system recovers with or without operator intervention.
- **Unrecoverable.** In unrecoverable failures, the system may need to be restarted.
- **Cosmetic.** These classes of failures cause only minor irritations, and do not lead to incorrect results. An example of a cosmetic failure is the case where the mouse button has to be clicked twice instead of once to invoke a given function through the graphical user interface.

Reliability growth models

A reliability growth model is a mathematical model of how software reliability improves as errors are detected and repaired. A reliability growth model can be used to predict when (or if at all) a particular level of reliability is likely to be attained. Thus, reliability growth modeling can be used to determine when to stop testing to attain a given reliability level. Although several different reliability growth models have been proposed, in this text we will discuss only two very simple reliability growth models.

Jelinski and Moranda Model

The simplest reliability growth model is a step function model where it is assumed that the reliability increases by a constant increment each time an error is detected and repaired. Such a model is shown in fig. 13.2. However, this simple model of reliability which implicitly assumes that all errors contribute equally to reliability growth, is highly unrealistic since it is already known that correction of different types of errors contribute differently to reliability growth.

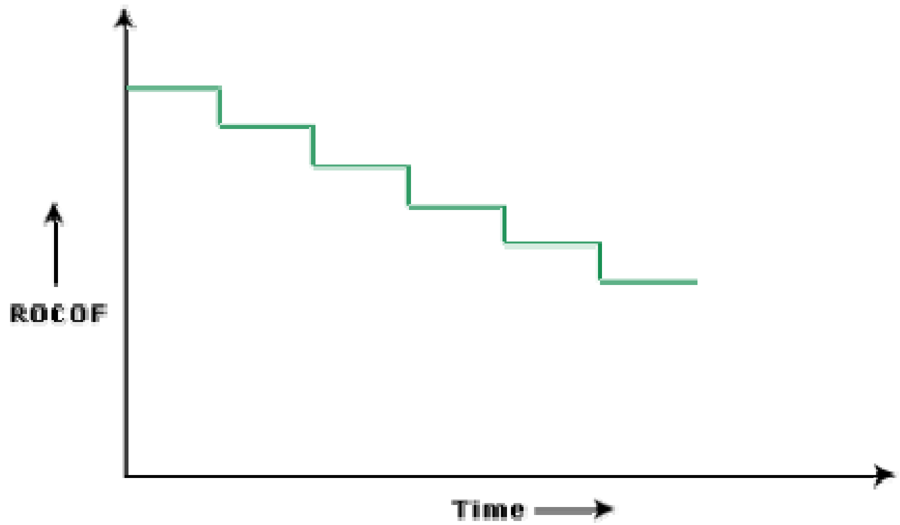


Fig. 13.2: Step function model of reliability growth

Littlewood and Verall's Model

This model allows for negative reliability growth to reflect the fact that when a repair is carried out, it may introduce additional errors. It also models the fact that as errors are repaired, the average improvement in reliability per repair decreases (Fig. 13.3). It treat's an error's contribution to reliability improvement to be an independent random variable having Gamma distribution. This distribution models the fact that error corrections with large contributions to reliability growth are removed first. This represents diminishing return as test continues.

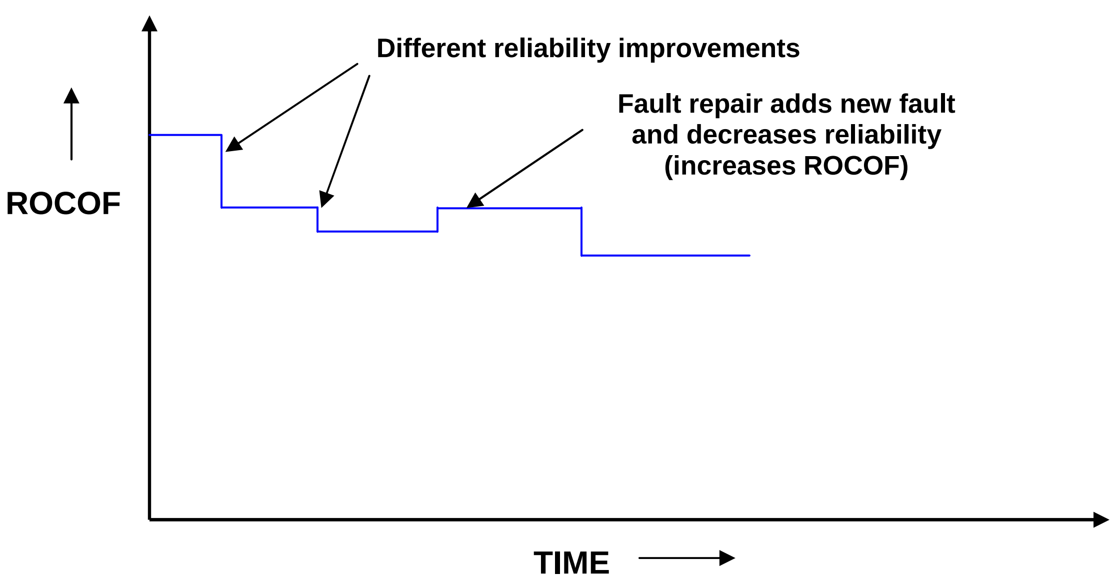


Fig. 13.3: Random-step function model of reliability growth

Statistical Testing and Software Quality Management

Specific Instructional Objectives

At the end of this lesson the student would be able to:

- Identify the primary objective of statistical testing.
- Define what is meant by the operation profile of a software product.
- Identify the steps in which statistical testing is performed on a software product.
- Identify the advantages and disadvantages of statistical testing.
- Identify the main quality factors of a software product.
- Explain what is meant by quality management system.
- Identify the phases over which quality management system has evolved in the last century.

Statistical testing

Statistical testing is a testing process whose objective is to determine the reliability of software products rather than discovering errors. Test cases are designed for statistical testing with an entirely different objective than those of conventional testing.

Operation profile

Different categories of users may use a software for different purposes. For example, a Librarian might use the library automation software to create member records, add books to the library, etc. whereas a library member might use to software to query about the availability of the book, or to issue and return books. Formally, the operation profile of a software can be defined as the probability distribution of the input of an average user. If the input to a number of classes {Ci} is divided, the probability value of a class represent the probability of an average user selecting his next input from this class. Thus, the operation profile assigns a probability value Pi to each input class Ci.

Steps in statistical testing

Statistical testing allows one to concentrate on testing those parts of the system that are most likely to be used. The first step of statistical testing is to determine the operation profile of the software. The next step is to generate a set of test data corresponding to the determined operation profile. The third step is to apply the test cases to the software and record the time between each failure. After a statistically significant number of failures have been observed, the reliability can be computed.

Advantages and disadvantages of statistical testing

Statistical testing allows one to concentrate on testing parts of the system that are most likely to be used. Therefore, it results in a system that the users to be more reliable (than actually it is!). Reliability estimation using statistical testing is more accurate compared to those of other methods such as ROCOF, POFOD etc. But it is not easy to perform statistical testing properly. There is no simple and repeatable way of defining operation profiles. Also it is very much cumbersome to generate test cases for statistical testing cause the number of test cases with which the system is to be tested should be statistically significant.

Software Quality

Traditionally, a quality product is defined in terms of its fitness of purpose. That is, a quality product does exactly what the users want it to do. For software products, fitness of purpose is usually interpreted in terms of satisfaction of the requirements laid down in the SRS document. Although “fitness of purpose” is a satisfactory definition of quality for many products such as a car, a table fan, a grinding machine, etc. – for software products, “fitness of purpose” is not a wholly satisfactory definition of quality. To give an example, consider a software product that is functionally correct. That is, it performs all functions as specified in the SRS document. But, has an almost unusable user interface. Even though it may be functionally correct, we cannot consider it to be a quality product. Another example may be that of a product which does everything that the users want but has an almost incomprehensible and unmaintainable code. Therefore, the traditional concept of quality as “fitness of purpose” for software products is not wholly satisfactory.

The modern view of a quality associates with a software product several quality factors such as the following:

- **Portability:** A software product is said to be portable, if it can be easily made to work in different operating system environments, in different machines, with other software products, etc.

- **Usability:** A software product has good usability, if different categories of users (i.e. both expert and novice users) can easily invoke the functions of the product.

- **Reusability:** A software product has good reusability, if different modules of the product can easily be reused to develop new products.

- **Correctness:** A software product is correct, if different requirements as specified in the SRS document have been correctly implemented.

- **Maintainability:** A software product is maintainable, if errors can be easily corrected as and when they show up, new functions can be easily added to the product, and the functionalities of the product can be easily modified, etc.

Software quality management system

A quality management system (often referred to as quality system) is the principal methodology used by organizations to ensure that the products they develop have the desired quality. A quality system consists of the following:

- **Managerial Structure and Individual Responsibilities.** A quality system is actually the responsibility of the organization as a whole. However, every organization has a separate quality department to perform several quality system activities. The quality system of an organization should have support of the top management. Without support for the quality system at a high level in a company, few members of staff will take the quality system seriously.

- **Quality System Activities.** The quality system activities encompass the following:
  - auditing of projects
  - review of the quality system
  - development of standards, procedures, and guidelines, etc.
  - production of reports for the top management summarizing the effectiveness of the quality system in the organization.

Evolution of quality management system

Quality systems have rapidly evolved over the last five decades. Prior to World War II, the usual method to produce quality products was to inspect the finished products to eliminate defective products. Since that time, quality systems of organizations have undergone through four stages of evolution as shown in the fig. 13.4. The initial product inspection method gave way to quality control (QC). Quality control focuses not only on detecting the defective products and eliminating them but also on determining the causes behind the defects. Thus, quality control aims at correcting the causes of errors and not just rejecting the products. The next breakthrough in quality systems was the development of quality assurance principles.

The basic premise of modern quality assurance is that if an organization's processes are good and are followed rigorously, then the products are bound to be of good quality. The modern quality paradigm includes guidance for recognizing, defining, analyzing, and improving the production process. Total quality management (TQM) advocates that the process followed by an organization must be continuously improved through process measurements.

TQM goes a step further than quality assurance and aims at continuous process improvement. TQM goes beyond documenting processes to optimizing them through redesign. A term related to TQM is Business Process Reengineering (BPR). BPR aims at reengineering the way business is carried out in an organization. From the above discussion it can be stated that over the years the quality paradigm has shifted from product assurance to process assurance (as shown in fig. 13.4).

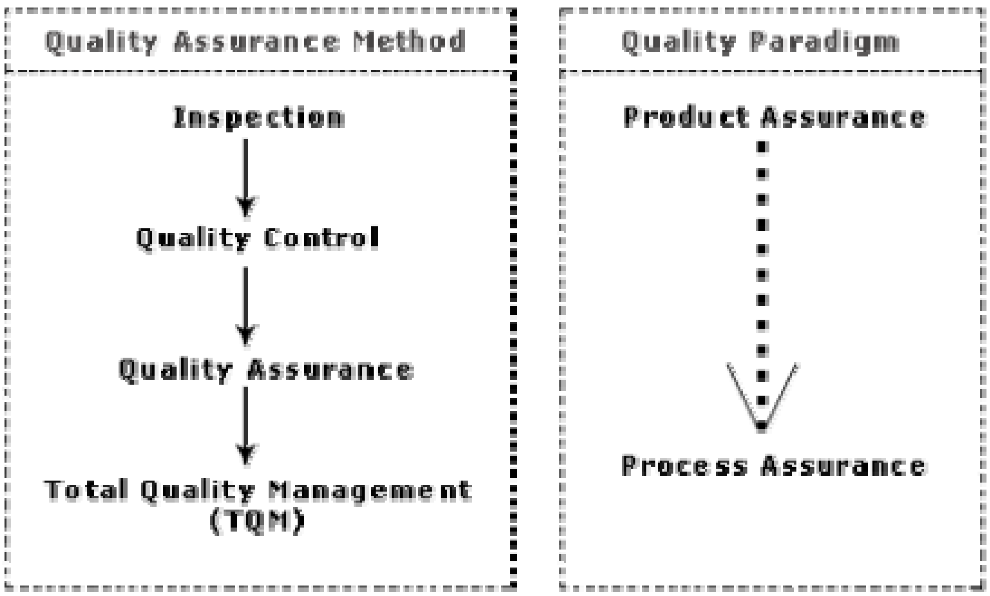


Fig. 13.4: Evolution of quality system and corresponding shift in the quality paradigm