BACHELOR OF COMPUTER APPLICATIONS

SEMESTER 5

DCA3103

SOFTWARE ENGINEERING

Unit 3

Software Reliability

Table of Contents

1. INTRODUCTION

For any software organization consistency or reliability is the significant vigorous attribute. End-users end up with lofty expenses if the software delivered is unreliable. Developers of defective or unreliable systems possibly will obtain a bad name for their quality and lose further business prospects in future. Software consistency or Reliability is generally defined as the possibility of operation with no failures for a particular time in a particular setting for a particular purpose. It is a kind of measure of how well a software system provides the services needed by the user. For example, software installed in an aircraft will be 99.99% reliable during an average flight of five hours. This means that a software failure of some kind will probably occur in one flight out of 10000. In this unit, we will study software reliability and software reliability metrics. We will also cover programming for reliability and software reuse.

1.1 Objectives

After studying this unit, you should be able to:

- ❖ *Define software reliability and differentiate between the terms fault, error and failure*
- ❖ *Explain software reliability metrics*
- ❖ *Describe programming for reliability*
- ❖ *Explain the benefits of software reuse*

2. SOFTWARE RELIABILITY

Reliability is usually defined as the probability of failure-free operation for a specified time in a specified environment for a specific purpose. Software reliability is a function of the number of failures experienced by a particular user of that software.

A software failure occurs when the software is executing. It is a situation in which the software does not deliver the service expected by the user.

Software failures are not the same as software faults although these terms are often used interchangeably. If you measure software reliability in one environment, you cannot assume that the reliability might be similar in another environment where the system is used differently.

For example, let's say that you measure the reliability of a word processor in an office environment where most users are uninterested in the operation of the software. They follow the instructions for its use and do not try to experiment with the system. If you measure software reliability in a university environment, then the reliability might be quite different.

Here students may explore the boundaries of the system and use the system in unexpected ways. These may result in system failures that did not occur in the more constrained office environment. Human perceptions and patterns of use are also significant.

For example, say a car has a fault in its wiper system that results in intermittent failures of the wipers to operate correctly in heavy rain. The reliability of that system as perceived by a driver depends on where they live and use the car.

A driver in a wet climate will probably be more affected by this failure than a driver in a dry climate. The wet climate driver perception will be that the system is unreliable, whereas the driver in a dry climate may never notice the problem.

In software reliability, it is helpful to distinguish between the terms fault, error and failure.

- **Failure:** An event that occurs at some point in time when the software does not deliver a service as expected by its user.

- **Error:** A mistake in software can lead to system behaviour that is unexpected by system users.
- **Fault:** A characteristic of a software system that can lead to a system error. For example, failure to initialize a variable could lead to that variable having the wrong value when it is used.

3. SOFTWARE RELIABILITY METRICS

Metrics which have been used for software reliability specification are shown in Table 3.1. The choice of which metric should be used depends on the type of system to which it applies and the requirements of the application domain. For some systems, it may be appropriate to use different reliability metrics for different sub-systems.

Table 3.1: Reliability metrics

Metric	Explanation
Probability of Failure on Demand (POFOD)	The probability that the system will fail when a service request is made. A POFOD of 0.01 means that one out of a thousand service requests may result in failure
Rate of Failure Occurrence (ROCOF)	The probability that the system will fail when a service request is made. A POFOD of 0.01 means that one out of a thousand service requests may result in failure
Mean Time to Failure (MTTF)	The average time between observed system failures. An MTTF of 500 means that one failure can be expected every 500-time unit.
Availability (AVAIL)	The probability that the system is available for use at a given time. Availability of 0.998 means that the system is likely to be available for 998 of every 1,000-time units.

In some cases, system users are most concerned about how often the system will fail, perhaps because there is a significant cost in restarting the system. In such cases, a metric the Mean Time to Failure (MTTF) or rate of failure occurrence metric base should be used. In other cases, a system must always meet a request for service because there is some cost in failing to deliver the service. The number of failures in some periods is less important. In those cases, a metric based on the Probability of Failure on Demand (POFOD) should be used. Finally, users or system operators may be most concerned when a service request is made the system should be readily available for service. They will incur some loss if the system is unavailable. Availability (AVAIL) considers the repair or restart time.

There are three kinds of measurement, which can be made when assessing the reliability of a system:

1) System failures are decided based on the inputs which are used to measure POFOD.
2) The time (or the number of transactions) between system failures is used to measure ROCOF and MTTF.
3) The elapsed repair or restart time when a system failure occurs. Given that the system must be continuously available, this is used to measure AVAIL.

Time is a factor in all of these reliability metrics. The appropriate time units must be chosen if measurements are to be meaningful. Time units, which may be used, are calendar time, the number of transactions or processor time.

In systems that spend much of their time waiting to respond to a service request, such as telephone switching systems, the time unit that should be used is processor time. If you use calendar time, then this includes the time when the system was doing nothing.

Calendar time is an appropriate time unit to use for systems that are in continuous operation. For example, monitoring systems such as alarm systems and other types of process control systems fall into this category.

Various types of software failures affect the software reliability of the system. Examples of different types of software failures are:

Failure Class	Description
Transient	This type of failure happens with specific inputs
Permanent	This type of failure happens with all inputs
Recoverable	The system can recover without operator intervention
Unrecoverable	Operator intervention needed to recover from failure
Non-corrupting	Failure does not corrupt the system state or data
Corrupting	Failure corrupts the system state or data

4. PROGRAMMING FOR RELIABILITY

In this, there is a general requirement for more reliable systems in all application domains. Customers expect their software to operate without failures and to be available when it is required. Improved programming techniques, better programming languages and better-quality management have led to very significant improvements in reliability for most software. However, for some systems, such as those, which control unattended machinery, these 'normal' techniques may not be enough to achieve the level of reliability required. In these cases, special programming techniques may be necessary to achieve the required reliability. Some of these techniques are seen in this chapter.

Reliability in a software system can be achieved using three strategies:

- **Fault avoidance:** This is the most important strategy, which applies to all types of systems. The design and implementation process should be organized to produce fault-free systems.

Example: a) Following coding standards and best practices to ensure clean and robust code. b) Applying defensive programming techniques, such as input validation and error handling, to prevent unexpected faults.

- **Fault tolerance:** In this, even though the system fails, facilities are provided in the software to continue the operations.

Example: a) Implementing redundancy in a distributed system by replicating data across multiple servers. If one server fails, the system can still operate using the replicas.

b) Utilizing load balancers to distribute incoming requests across multiple servers. If one server becomes overloaded or fails, the load balancer redirects traffic to other healthy servers.

- **Fault detection:** In this, the faults are identified before the software is put into operation.

Example: a) Performing comprehensive testing, including unit testing, integration testing, and system testing, to identify and fix faults before deploying the software.

b) Utilizing monitoring and logging tools to track system behavior and identify any abnormal patterns or errors that may indicate faults.

4.1 Fault Avoidance

A good software process should be oriented towards fault avoidance rather than fault detection and removal. It should have the objective of developing fault-free software. Fault-free software means software, which conforms to its specification. Of course, there may be errors in the specification or it may not reflect the real needs of the user so fault-free software does not necessarily mean that the software will always behave as the user wants.

Fault avoidance and the development of fault-free software rely on:

1. The availability of an exact system specification, which is a clear description of what must be implemented.
2. The adoption of an organizational quality philosophy in which quality is the driver of the software process. Programmers should expect to write bug-free programs.
3. The adoption of an approach to software design and implementation which is based on information hiding and encapsulation and encourages the production of readable programs.
4. The use of a strongly typed program language so that possible errors are detected by the language compiler.
5. Restriction on the use of programming constructs, such as pointers, which are inherently error-prone.

Achieving fault-free software is virtually impossible if low-level programming languages with limited type-checking are used for program development. Figure 3.1 shows an increasing cost of the residual fault of removal.

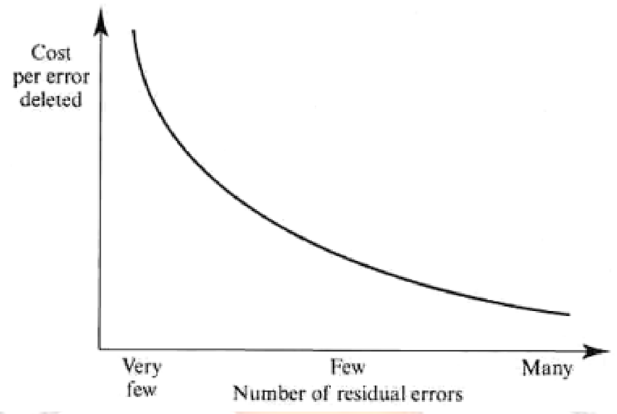

Fig 3.1: The increasing cost of residual fault removal.

We must be realistic and accept that human errors will always occur. Faults may remain in the software after development. Therefore, the development process must include a validation phase, which checks the developed software for the presence of faults. This validation phase is usually very expensive. As faults are removed from a program, the cost of finding and removing remaining faults tends to rise exponentially. As the software becomes more reliable, more and more testing is required to find fewer and fewer faults.

Structured programming and error avoidance

Structured programming is programming using only while loops and if statements as control constructs and designing using a top-down approach. This programming does not use any go-to statements. The adoption of structured programming was an important milestone in the development of software engineering because it was the first step away from an undisciplined approach to software development.

Go to the statement was an inherently error-prone programming construct. The disciplined use of control structures forces programmers to think carefully about their programs. Hence, they are less likely to make mistakes during development. Structured programming means programs can be read sequentially and are therefore easier to understand and inspect. However, avoiding unsafe control statements is only the first step in programming for reliability.

If the use of these constructs is minimized, then the faults are less likely to be introduced into programs, these constructs include:

1) ***Floating-point numbers:*** Floating-point numbers are inherently imprecise. They present a particular problem when they are compared because representation imprecision may lead to invalid comparisons. Fixed-point numbers, where a number is represented to a given number of decimal places, are safer as exact comparisons are possible.
2) ***Pointer:*** Pointer refers to a location in the memory. They cause errors because they allow 'aliasing'. This means the same memory location can be accessed using different names. Aliasing makes programs harder to understand so that errors are more difficult to find. However, it is often impractical to avoid the use of pointers.
3) ***Dynamic memory allocation:*** Program memory is allocated at run-time rather than compile-time. The danger with this is that the memory may not be de-allocated so the system eventually runs out of available memory. This can be a very intelligent type of error to detect as the system may run successfully for a long time before the problem occurs.
4) ***Parallelism:*** Parallelism is dangerous because of the difficulties of predicting the effects of timing interactions between parallel processes. Timing problems generally cannot be identified by program checks and the irregular combination of circumstances, which might affect a timing problem. Parallelism is necessary but its use should be carefully controlled to minimize inter-process dependencies. Programming language facilities, such as Ada tasks, help avoid some of the problems of parallelism as the compiler can detect some kinds of programming errors.
5) ***Recursion:*** The process of function calling itself repeatedly is known as recursion. Its use can result in very concise programs but it can be difficult to follow the logic of recursive programs. Errors in using recursion may result in the allocation of the system's memory as temporary stack variables are created.
6) ***Interrupts:*** Interrupts are a means of forcing control to transfer to a section of code irrespective of the code currently executing. The problem with the interrupt is, it might affect a complex operation to be terminated.

4.2 Fault Tolerance

In a fault-tolerance system, the operations are continued even after system failures have occurred, even though there is a failure in some of its components the system enables to continue operating properly.

Some of the features of fault tolerance are seen below:

1. ***Failure detection:*** This will detect the failure of the system. Failure may be due to hardware or software.
2. ***Damage assessment:*** This will identify the damaged parts of the system, which have been affected by the failure.
3. ***Fault recovery:*** Fault recovery is a process that involves restoring an error state to an error-free state.

Example: *a) Rollback and transaction log replay:* Database systems provide the ability to roll back an unfinished transaction and replay the logged operations to return the database to a consistent state when a failure occurs during a transaction.

b) Rebooting or restarting: A system or component may occasionally need to be rebooted or restarted to recover from a problem. To get everything back to normal, this may need restarting a server, services, or processes.

c) Switchover and failover: In a redundant system, if a primary component fails, fault recovery can involve switching over to a secondary component or activating a failover system to take over the workload and ensure uninterrupted service.

4. **Fault repair:** This involves modifying the system so that the fault does not recur. In many cases, software failures are transient, due to a peculiar combination of system inputs. No repair is necessary and normal processing can resume immediately after fault recovery.

Example: *a) Algorithm or design improvements:* Making changes to algorithms or system design to improve fault tolerance, error detection, or recovery capabilities.

b) Infrastructure changes: Modifying the hardware or network infrastructure to address hardware failures, bottlenecks, or resource limitations that may lead to faults.

c) Code or configuration updates: Modifying the software code or configuration settings to fix bugs, address vulnerabilities, or enhance error handling mechanisms.

If there are no faults in the system, there would not seem to be any chance of system failure. However, 'fault-free' does not mean 'failure-free'. It can only mean that the program corresponds to its specification. The specification may contain errors or omissions or incorrect assumptions about the system's environment. We can never demonstrate that the system is completely fault-free. In systems that have the highest reliability and availability requirements, you need to use redundant and diverse approaches of fault avoidance and fault tolerance.

4.3 Fault Detection

The use of verification and validation techniques increases the chances that faults will be detected and removed before the system is used. Systematic testing and debugging are an example of fault-detection techniques.

5. SOFTWARE REUSE

Software reuse is the process of creating software systems from existing software rather than building them from scratch. The design process in most engineering disciplines is based on the reuse of existing systems or components.

For example, mechanical or electrical engineers do not specify a design in which every component has to be manufactured especially. They base their design on components that have been tried and tested in other systems.

These are not just small components but include major sub-systems such as engines, condensers or turbines.

Software products are expensive; therefore, software project managers are always worried about the high cost of software development and are desperately looking for way-outs to cut development costs. A possible way to reduce development costs is to reuse parts from previously developed software. In addition to reduced development cost and time, reuse also leads to a higher quality of the developed products since the reusable components are ensured to have high quality.

Reuse-based software engineering is a comparable software engineering strategy where the development process is geared toward reusing existing software. Although the benefits of reuse have been recognized for many years, it is only in the past 10 years that there has been a gradual transition from original software development to reuse-based development. The move to reuse-based development has been in response to demands for lower software production and maintenance costs, fast delivery of systems and increased software quality. More and more companies see their software as a valuable asset and promote reuse to increase their return on software investments.

Apart from libraries such as window system libraries, there is no common base of reusable software components, which is known by all software engineers. However, this situation is slowly changing. We need to reuse our software assets rather than redevelop the same software again and again.

Demands for lower software production and maintenance costs along with increased quality can only be met by widespread and systematic software reuse. The reuse of a program is just not meant to reuse existing code or modules. Designs and requirements are likely reuse. Reusing intangible products of the software development process has latent gains, such as requirements, which would be bigger than those from reusing module code.

Reuse–based software engineering is an approach to development that tries to maximize the reuse of existing software. The software units that are reused may be of radically different sizes.

For example, the reuse of software can consider at several different levels:

1) ***Application System Reuse:*** The whole of an application system may be reused. The key problem here is ensuring that the software is portable; it should execute on several different platforms.
2) ***Sub-system Reuse:*** Major sub-systems of an application may be reused. For example, a pattern-matching system developed as part of a text processing system may be reused in a database management system.
3) ***Module or Object Reuse:*** Components of a system representing a collection of functions may be reused. For example, an Ada package or a C++ object implementing a binary tree may be reused in different applications.

4) ***Function Reuse:*** Software components, which implement a single function, such as a mathematical function, may be reused.

One of the ***advantages*** of software reuse is cost reduction. Some of the ***benefits*** *of reusing software are given below:*

- ***Reduced Process Risk:*** The cost of existing software is already known, while the costs of development are always a matter of judgement. This is an important factor for project management because it reduces the margin of error in project cost estimation. This is particularly true when relatively large software components such as sub-systems are reused.
- ***Effective Use of Specialists:*** Instead of doing the same work over and over, these application specialists can develop reusable software that encapsulates their knowledge.
- ***Standards Compliance:*** Some standards, such as user interface standards, can be implemented as a set of standard reusable components.

For example, in a user interface, if menus are implemented using reusable components, the same menu format is applied to all applications like (word, PowerPoint, Excel and many more) so that a standard or common user interface is maintained and users become more familiar and make fewer mistakes while using the menus.

- ***Accelerated Development:*** Bringing a system to market as early as possible is often more important than overall development costs. Reusing software can speed up system production because both development and validation time should be reduced.

However, there are also costs and problems associated with reuse. In particular, Systematic reuse does not just happen, it must be planned and introduced through an organisation. Companies such as Hewlett-Packard have also been more popular in their reuse programs.

Some of the problems with software reuse are seen below:

- *Increased Maintenance Costs:* If the source code of a reused software system or component is not available then maintenance costs may be increased because the reused elements of the system may become increasingly incompatible with system changes.

- *Lack of Tool Support:* CASE toolsets may not support development with reuse. It may be difficult or impossible to integrate these tools with a component library system. The software process assumed by these tools may not take reuse into account.
- *Rewrite Software:* Some software engineers prefer to rewrite components because they believe they can improve on them. This is partly to do with trust and partly to do with the fact that writing original software is seen as more challenging than reusing other people's software.
- *Creating and Maintaining a Component Library:* Populating a reusable component library and ensuring the software developers can use this library can be expensive. The current techniques for classifying, cataloguing and retrieving software components are immature.
- *Finding, Understanding and Adapting Reusable Components:* Software components have to be discovered in a library, and sometimes adapted to work in a new environment. Engineers must be reasonably confident in finding a component in the library before they will include a component search as part of their normal development process.

6. SUMMARY

- Reliability is usually defined as the probability of failure-free operation for a specified time in a specified environment for a specific purpose.
- In software reliability, there are various reliability metrics such as Probability of Failure on Demand (POFOD), Rate Of Failure Occurrence (ROCOF), Mean Time To Failure (MTTF), and Availability (AVAIL).
- various types of software failures affect the software reliability of the system. Examples of different types of software failures are: Transient, permanent, recoverable, unrecoverable, non-corrupting and corrupting.
- In programming for reliability, customers expect their software to operate without failures and to be available when it is required.
- Reliability in a software system can be achieved using three strategies such as fault avoidance, fault tolerance and fault detection
- Structured programming is programming using only while loops and if statements as control constructs and designing using a top-down approach. This programming does not use any go-to statements.
- If the use of these constructs is minimized, then the faults are less likely to be introduced into programs, these constructs include floating-point numbers, pointers, dynamic memory allocation, parallelism, recursion, and interrupts.
- Some of the features of fault tolerance are failure detection, damage assessment, fault recovery and fault repair.
- Software reuse is the process of creating software systems from existing software rather than building them from scratch

7. SELF-ASSESSMENT QUESTIONS

SELF-ASSESSMENT QUESTIONS – 1

1. ____________________ is defined as the probability of failure-free operation for a specified time in a specified environment for a specific purpose.
2. MTTF stands for __________________.
3. In which type of failure class, failure happens with specific inputs?
4. In ___________________ strategy even though the system fails, facilities are provided in the software to continue the operations.
5. In ___________________ strategy, the faults are identified before the software is put into operation.
6. Pointer refers to a location in the ____________________.
7. _____________________ is the process of creating software systems from existing software rather than building them from scratch.
8. Mention any two benefits of reusing the software.

8. SELF-ASSESSMENT ANSWERS

1. Reliability
2. Mean Time to Failure
3. Transient
4. Fault tolerance
5. Fault Detection
6. Memory
7. Software reuse
8. Reduced process risk and effective use of specialists.

9. TERMINAL QUESTIONS

1. What is software reliability? Explain.
2. Explain different software reliability metrics.
3. How reliability in a software system can be achieved? Explain.
4. What are the different levels of software reuse?

10. TERMINAL ANSWERS

1. Reliability is usually defined as the probability of failure-free operation for a specified time in a specified environment for a specific purpose. Software reliability is a function of the number of failures experienced by a particular user of that software. (Refer to section 3.2).
2. In software reliability, there are various reliability metrics such as Probability of Failure on Demand, Rate of Failure Occurrence, Mean Time to Failure, and Availability. (Refer to section 3.3).
3. In programming for reliability, customers expect their software to operate without failures and to be available when it is required. Reliability in a software system can be achieved using three strategies such as fault avoidance, fault tolerance and fault detection. (Refer to section 3.4).
4. The reuse of software can consider at several different levels such as Application System Reuse, Sub-system Reuse, Module or Object Reuse, and Function Reuse. (Refer to section 3.5).