Preamble

This material is released under a Creative Commons License agreement http://creativecommons.org/licenses/by/2.0/uk/ In informal terms we, the authors, give you permission to do most things with the material provided you:

a) Acknowledge us as the authors of the original material, and retain the acknowledgement in copy of the material, and in any of your material based on ours,
b) Do not pretend you wrote it,
c) Do identify any changes you make as your work, not ours.

The material includes opinions of the authors, and may be different from work published by others. We have quoted and referenced work from third parties, and in a few cases have knowingly used or adapted ideas or content from third parties. Where appropriate we ask for permission from the third parties.

However we make mistakes, as most people do, and may have made mistakes in this work. If you spot something you feel is wrong, or simply something you feel could be improved please let us know via this email address nft.introduction@commercetest.com

We are Stuart Reid and Julian Harty, and are the authors of this material: **An introductory course on non-functional software testing**. If you wish to create derived works you need include the following acknowledgement: at the start and end of the derived works: **This work is based on original material by Stuart Reid and Julian Harty.** We like to know whether our work is being used so please tell us how you're using the content.

Finally, this work is still in a draft form so you may find inconsistencies, gaps, incomplete content, etc. We hope you will find the content useful anyway and we are very happy to receive suggestions for improvement to the email address mentioned above.

Software Dependability

Definition of
Software Dependability

- The collective term used to describe the availability performance and its influencing factors: reliability performance, maintainability performance and maintainability support performance.
  - IEC 60050(191):1990 International Electrotechnical Vocabulary

1. As can be seen from the 'official' definition above, software dependability comprises a mix of three attributes: availability, reliability and maintainability. These three attributes are closely related, but to understand this relationship, each attribute must first be clearly understood in isolation.

Definition of
Software Dependability

Common Usage

The trustworthiness of a computer system such that reliance can be justifiably placed on the service it delivers.

2. A more popular definition of software dependability is provided above. Most would agree that something we trust and rely on is considered to be dependable.

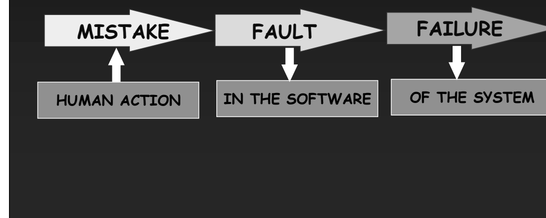
Software Dependability

- Reliability
- Maintainability
- Availability
- Recoverability

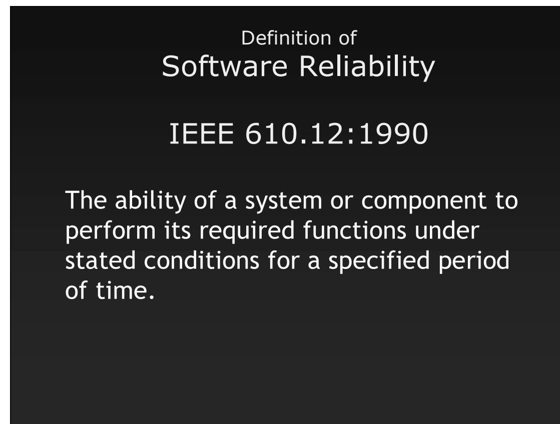
3. We shall consider each of the attributes that make up dependability in turn and then show how they are related to each other before concluding this section by briefly considering recoverability (and recovery testing) due to its close relationship to availability.

4. The first dependability attribute to be considered will be software reliability. Software reliability is especially relevant for safety-related systems, where the consequence of system failure may be injury or death, although it is also used for non-safety-related software.

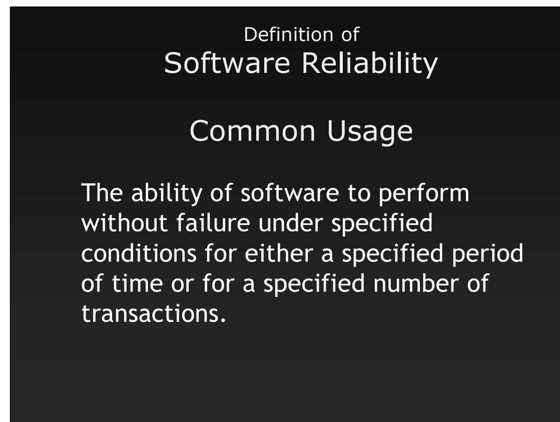
Mistakes, Faults and Failures



5. There are a number of definitions of software reliability, but as they are all based on software failures, we must first explain how a software system fails. Initially a person makes a mistake (for whatever reason, but perhaps because they are stressed or rushing) and this leads to there being a fault in the software (a very simple example could be that a decimal point has been misplaced when typing in a numerical value). Finally, if the right set of conditions coincide to cause the fault to be activated then the system fails.



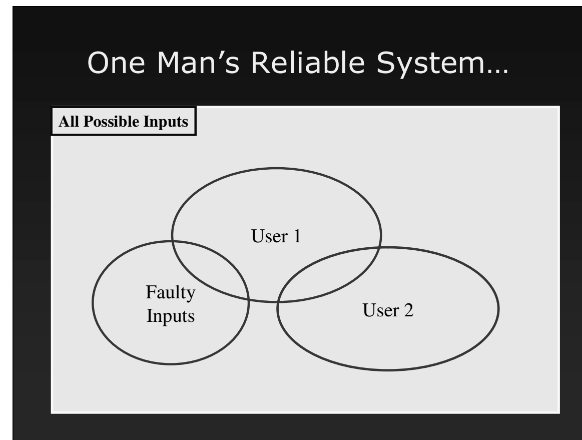
6. As shown above, the IEEE definition of software reliability, like most definitions of reliability, concentrates on the idea of the system running without failure for a *period of time*. For software that does not run continuously (e.g. demand-driven software) this is not appropriate, so we also need to consider the concept of defining reliability in terms of the number of transactions performed without failure.



7. A more 'complete' definition of software reliability is shown above. This works for both continuous and demand-driven software.

8. From the above definitions it can be seen that reliability is highly dependent on the "stated/specified conditions". These conditions include factors such as the environment the software is to run in (e.g. what operating system, what processor, which other systems it will be communicating with), and how the software is expected to be used. Thus when we perform reliability testing we attempt to set up tests that are as representative of the expected operational use of the software as possible in terms of both the environment and the expected use. The expected operational environment and usage is normally described by what is known as the 'operational profile' of the software.

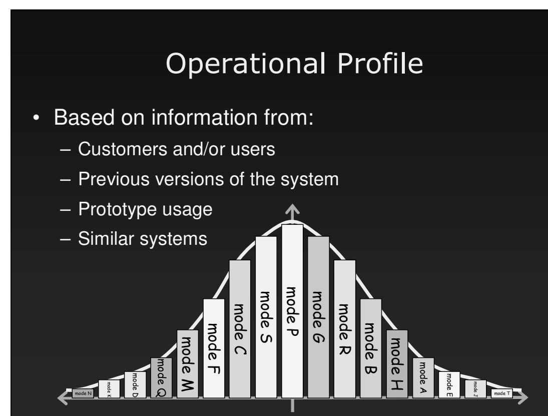
9. Reliability testing is the closest of all the non-functional testing techniques to functional testing. The quality characteristic of reliability is simply based on the rate of failure of the software to meet its functional requirements. The major difference between reliability testing and functional (defect) testing is that with reliability testing the test inputs aim to mimic real life (i.e. the operational profile) rather than being aimed at exercising specific functions or finding specific types of fault.



10. If the operational profile is not correctly defined for the software, then the results of reliability testing can be seriously flawed. For instance consider the situation above where all possible inputs for a software system are shown in the box. As with all software there will be a set of inputs that if executed by the software will cause a failure – these are labelled as the 'Faulty Inputs' (note that these do not correspond to inputs where the user has made a mistake, but rather to those inputs that cause a latent fault to lead to system failure). The sets of inputs of two quite different users are also shown as 'User 1' and 'User 2'. The 'User 1' set overlaps with the 'Faulty Inputs' set and so occasionally User 1 will find that when they select an input that is also in the 'Faulty Inputs' set then the software fails. User 2, however, in this (very unlikely) example, finds that as far as they are concerned the software is completely reliable and never fails because none of their inputs overlap with the 'Faulty Inputs' set.

11. Thus we have two users of the same software with completely different impressions of its reliability. One of the most important aspects when trying to measure and predict software reliability is determining the correct operational profile (how users interact with the system) for *all* the users of the system.

12. This operational profile should be defined as part of the reliability requirements for the software. If it is not clearly stated then it is possible for a software development organisation to carefully select an operational profile at the end of development that they know from their own testing will give an especially low number of failures, but might be completely unrepresentative of the future operational use of the software.



13. There are many ways of representing an operational profile, but it is often the case that the operational profiles used for software reliability simply consider different modes of use for the

system, and provide the relative probability of each mode being used, as shown above. A more thorough operational profile would also consider who the different users might be and the overall environment in which the software is expected to operate.

Probability of Failure on Demand (POFOD)

- The likelihood that the system fails when a demand is made of it.
- POFOD = 0.01 (or 1%) means that at least 99 out of every 100 requests must be dealt with successfully.
- The Sizewell B Primary Protection System has a requirement of POFOD = 0.001 (although the original requirement before it was measured was POFOD = 0.0001!).

14. Reliability can be directly stated as a probability; however, this probability is not always used in requirements specifications. For demand-driven systems a metric of the probability of failure on demand (POFOD) is often used as shown above both as a probability and a percentage.

Mean Time to Failure (MTTF)

- The time to the next system failure.
- MTTF = 10^3 hours means that the average time to the next failure is no less than 1000 hours.
- The Flight Control Software for the Airbus A320/340/etc. and Boeing 777 has a requirement of MTTF of 10^9 hours (approx. 114,000 years!!!)
  - this is not measurable given current technology.

15. For continuously-running systems then a metric such as the mean time to failure (MTTF), shown above, may be most appropriate, as shown above.

Definition of

Software Reliability

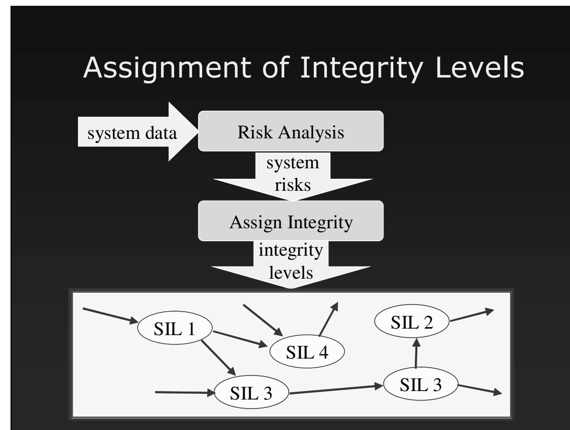
Common Usage

The ability of software to perform without failure under specified conditions for either a specified period of time or for a specified number of transactions.

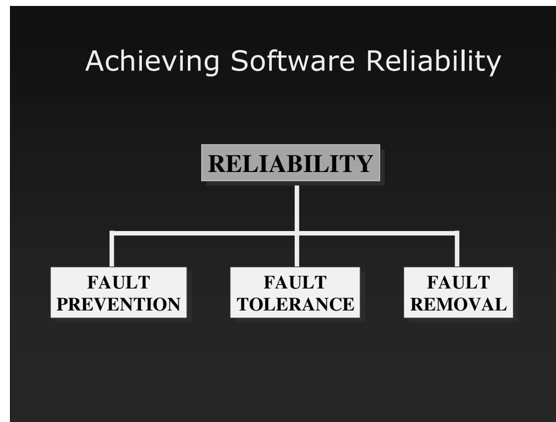
16. Given the definition of reliability (repeated above) it can be seen that when software reliability requirements are stated it is not enough to simply state the required time to failure or the number of acceptable failures in a time period and the 'specified conditions' in an operational profile. What is meant by a failure also has to be agreed. For instance, it would not usually be sensible to consider a simple spelling mistake in an on-screen message as a failure in the same way as a complete system crash.

17. For large systems we might find that a number of different metrics may be required to specify the reliability for each of its component subsystems separately. For instance, a system may have two functions; one, a monitoring routine, that runs continuously, and a second that runs only when an emergency situation is indicated. This could lead to the use of a MTTF for the monitoring routine, and a POFOD for the emergency function.

18. The assignment of values for reliability to the various component parts of software systems is done in a number of ways. Informal approaches are possible, and experience from previous systems can play a part, but generally if it is considered relevant to specify reliability then the systems tend to be of relatively high integrity, and this normally means that a more formal approach is taken. In some situations the assignment is determined by contractual agreement, with the customer deciding the required level. In others the assignment is determined analytically by considering the possible system failures and their consequences to identify commensurate reliability requirements.

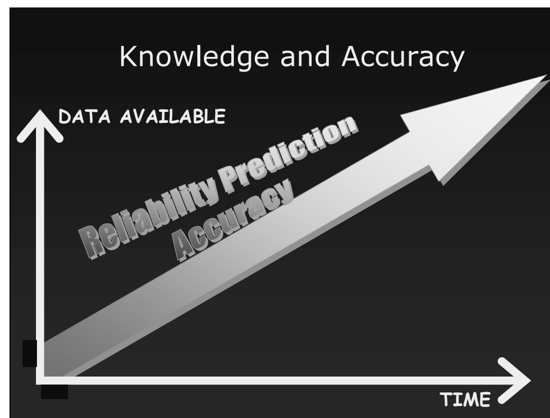


19. When software is being developed as part of a safety-related system then it nearly always has to comply with a regulatory standard that defines the necessary level of reliability to be achieved for given levels of integrity. These software integrity levels (SILs) are normally derived from a risk analysis of the software system as shown above. Safety-related standards, such as DO-178B which is used by most avionics manufacturers, typically include four or five levels of integrity, ranging from a level where no risk is associated with failure to a level where many people will die as a result of failure.

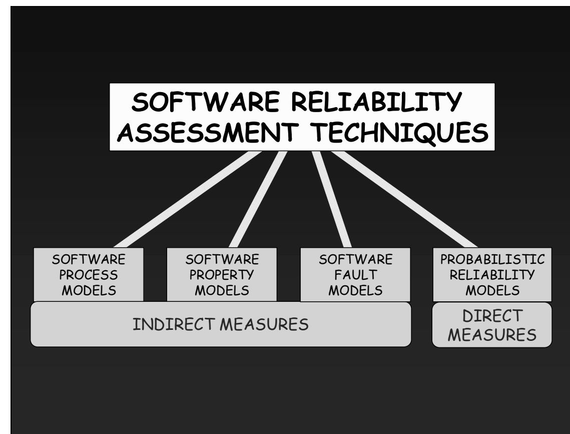


20. Software reliability is generally achieved by using combinations of three complementary approaches, as shown above. These three approaches are fault prevention (using good software development practices to 'ensure' a reliable product), fault removal (the detection and subsequent removal of faults from the software, normally known as verification, validation and testing) and fault tolerance (the inclusion of redundancy in the software to make it tolerant of local failures).

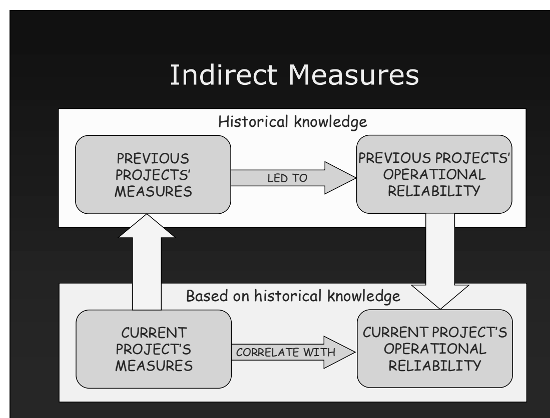
21. Where fault tolerance is used to achieve a required level of reliability, the extra software used to achieve redundancy means that there are potential conflicts with other quality characteristics such as performance response times and maintainability. Response times are typically compromised due to the extra processing that is needed to monitor for potential failures in order that recovery can be instigated. Maintainability is often reduced due to the extra complexity introduced by a fault tolerant design.



22. Given a reliability requirement for software, it should be possible to measure or test whether the requirement has been achieved. It would also be *useful* to be able to predict future software reliability, ideally from as early in development as possible. There are a number of ways of both measuring and predicting software reliability, and they tend to increase in accuracy the later they are applied in the life cycle.



23. Until the software system is fully integrated it is only possible to take indirect measures from which to predict future reliability. These measures are generally based on three different attributes of the development: the software development process, the software properties (its structure, language, etc.), and the faults detected in the software. Once an integrated system is available then it can be tested directly and probability-based predictions can be made about the software's reliability.



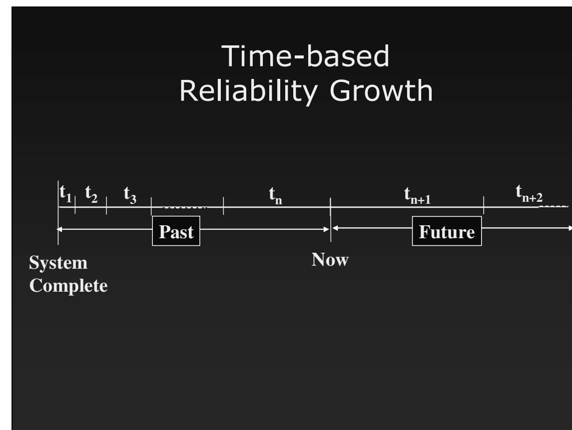
24. All three approaches using indirect measures are based on comparing measures from the current project with measures from previous projects as shown above. It is assumed that the relationship between these indirect measures on previous projects and their subsequent reliability will also be applicable to the current project, thus allowing future reliability to be inferred. Obviously any correlation between indirect measures and future reliability can only be determined if sufficient data is available from previous projects, highlighting the necessity of maintaining a database of metrics data on software projects.

25. Given that a software reliability requirement is specified as an indirect measure, then it should be relatively easy to test whether that requirement has been achieved. The two main techniques used to test whether indirect measures are achieved, or not, are reviews and static analysis. Let's consider two typical examples. First, imagine that the maximum size of individual code modules has been specified as 200 lines of code. It is then a simple matter to record the size of each module as it is submitted through the project's configuration management tool and raise an alert if any of them exceed the set limit. Second, imagine that certain design standards have been set (perhaps based on the coupling and cohesion of the design). In this scenario, it would be the responsibility of those taking part in design reviews to identify whether the set design standards had been met, or not.

26. The more direct means of measuring or testing reliability are failure-based and come from executing the complete system in an environment that is as similar to the operational environment as possible. We shall consider the traditional approach of reliability growth modelling, although there is an alternative approach based on statistical testing using Markov models of the software (state diagrams with probabilities assigned to the transitions between states), but this is rather specialised and only applicable to certain types of software.

27. With traditional software reliability testing, it is assumed that software reliability generally increases with time (as the faults are removed) and so we use mathematical models known as 'reliability growth models' to measure and predict reliability. These models may be time-based or transaction-based. Before reliability growth testing starts, the software under test should have satisfied the criteria for functional defect testing (and normally any other non-functional testing requirements).

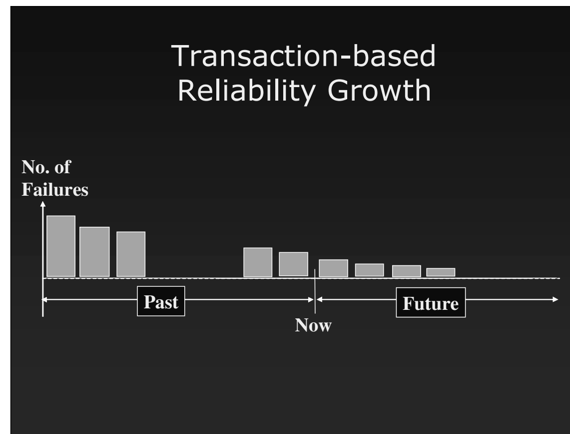
28. For both forms of reliability growth testing, the test inputs are normally generated using automated tools in a pseudo-random manner based on the operational profile of the software under test. The tests should also be run in an environment as close as possible to the expected operational environment.



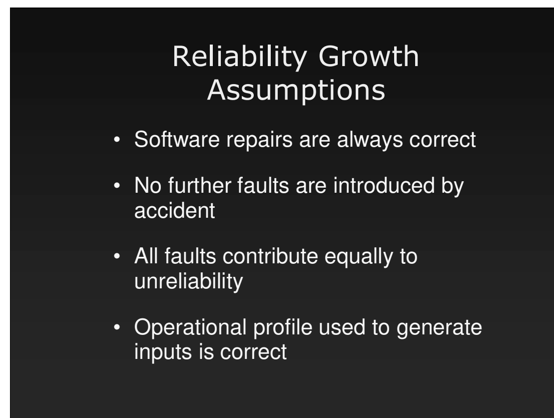
29. With time-based reliability growth testing, testing starts at the 'System Complete' point on the diagram and continues until a failure is detected (after period t_1 on the diagram). At this point the fault that led to the failure is determined and removed, before testing recommences. In an ideal world the software should run for longer before failure on this next run (period t_2) as there is now one less fault in the software to cause a failure. Once again the failure-causing fault is removed and testing restarts. This cycle of running for a measured time period, followed by the identification and removal of the failure-causing fault continues with the time period between failures always increasing (in ideal, but unrealistic, conditions).

30. Given a large enough 'set of times' between failures, which *should* be increasing in a predictable manner, it should be possible to forecast the future set of time periods. In fact, we are only really interested in the next time period, as if this is greater than that required by the reliability requirement for the software ('Now' on the diagram above), we stop testing as we can now say that we believe that the software meets (or exceeds) the reliability requirement.

31. The model so far presented relies on several assumptions, but before considering how we manage these, let's first consider how transaction-based reliability growth testing differs from its time-based counterpart.



32. In the transaction-based model a large set number of transactions is decided on (say perhaps a million) and this number of transactions are pseudo-randomly generated based on the operational profile. A number of failures will occur when these transactions are run and for each failure the corresponding fault is identified and removed from the software. Another set of transactions is then created and executed. The number of failures in each successive set *should* be decreasing as reliability increases. In the same way as for the time-based approach, at some point it should be possible to predict that the required level of reliability has been achieved and the system is then released into operation.

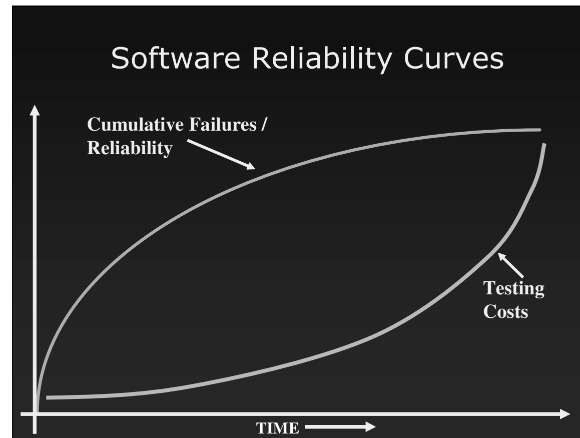


33. Basic reliability growth models work on a set of idealistic assumptions. These include: software faults that cause failures are immediately fixed, correctly, with no side effects; all faults contribute an equal likelihood to system failure; and all failures contribute an equal amount to unreliability. Obviously in the real world the chance of certain faults causing a failure is very rare, some failures are more serious than others, and debugging is not performed perfectly.

34. At present there is no single reliability growth model that can be recommended for all applications. So, the failure data produced as a result of the statistical testing of the software application is compared with a number of operational systems' reliability growth models and the model providing the 'best fit' is selected. Software tools are available that can carry this out automatically. All the models are based on the (possibly hopeful) assumption that the software's operational profile does not change.

35. It should be noted that reliability growth models are not suitable for predicting the very high levels of reliability required for systems such as safety-critical avionics software (e.g. one

failure in 10^9 hours or 114,000 years). This is because it would obviously require a prohibitive amount of testing time to produce enough data for the model to be able to predict that the next time to failure was so large. Normally the upper limit for reliability growth models is predictions of 10^6 hours.



36. The Software Reliability Curves shown above mirror a typical reliability growth model. They show reliability increasing with time, as the cumulative number of failures gradually levels off. Note that it also shows that to achieve ever higher reliability the testing costs rise ever higher.

Definition of
Software Maintainability

IEEE 610.12:1990

The ease with which a software system or component can be modified to correct faults, improve performance, or other attributes, or adapt to a changed environment.

37. When software is not completely reliable (as is always the case), it contains faults that, when discovered, often need to be removed. As can be seen from the definition above, software maintainability is partly concerned with how easy it is to remove these faults. It should be noted, however, that fault removal is typically only a small part of the overall maintenance burden. The majority of maintenance tasks are concerned with implementing new requirements and adapting to new environments, such as a new operating system.

Specification of Software Maintainability

- Mean Time To Repair (MTTR)
  - the average time taken to implement a change and restore the system to working order
- But, MTTR is very difficult to predict as we do not know in advance which repairs will need to be undertaken
  - otherwise we would repair them now

38. If we limit software maintenance to simply fixing faults, them the specification of software maintainability can be specified as a mean time to repair (MTTR), as shown above. As we have already seen, however, maintenance is not just fixing faults. A more generic measure would be a mean time to modify (MTTM), which could be measured in hours, days or weeks depending on the scale of the modification.

39. In practice, these measures are of limited value for two main reasons. First, for most software systems we do not know what changes are going to be required in the future (there are occasional examples where scheduled changes are known in advance). Second, the scale of the changes can vary dramatically. One change might require a single line of code change, while another might require thousands of new modules to be added to the system, with all the corresponding change management, integration, testing, etc.

40. Both MTTR and MTTM are direct measures of the maintainability of software, but have major drawbacks as mentioned above. Many organisations use indirect measures instead. These indirect measures work in the same way as for software reliability - by assuming that measures of certain attributes from the current project can be used to predict its future maintainability. You may remember that the main approaches to testing whether indirect requirements are met are reviews and static analysis.

Indirect Measures of Software Maintainability

- Software design method
- Languages and operating systems
- Design and coding standards
  - modularity
  - documentation
  - structured coding techniques
  - code readability
    - comments (internal documentation)
- Test case suites and regression testing

41. Shown above are a number of typical indirect measures that are considered by many to have an effect on the maintainability of software. The software design method to be on the project could be specified for two main reasons in terms of maintainability. First, it may be that an object-oriented method is specified instead of another approach as object-oriented methods are

popularly thought to lead to easier modification. Second, a customer may specify a particular design method to ease their future maintenance as by using the same design method on all projects it means they can train their maintainers on just that one method. Similarly, certain programming languages and operating systems are thought to lead to easier maintenance. It is certainly inarguable that modifying software written in high order languages, such as Java and C++ is far easier than making changes to systems written in machine code.

Obfuscated C Example

```
extern int
                                                      errno
                                                        ;char
                                                           grrr
                        ;main(                              r,
  argv, argc )          int    argc                          ,
   r       ;          char *argv[];{int                    P( );
#define x  int i,     j,cc[4];printf("     choo choo\n"     ) ;
x  ;if    (P(  !      i              )       |  cc[  !      j ]
&  P(j    )>2  ?      j              :       i  ){*  argv[i++ +!-i]
;              for    (i=            0;;    i++                   );
_exit(argv[argc- 2    / cc[1*argc]|-1<<4 ]    ) ;printf("%d",P(""));}}
  P  (    a  )   char a   ;  {    a  ;   while(    a  >      "  B   "
  /* -    by E            ricM    arsh             all-      */);    }
```

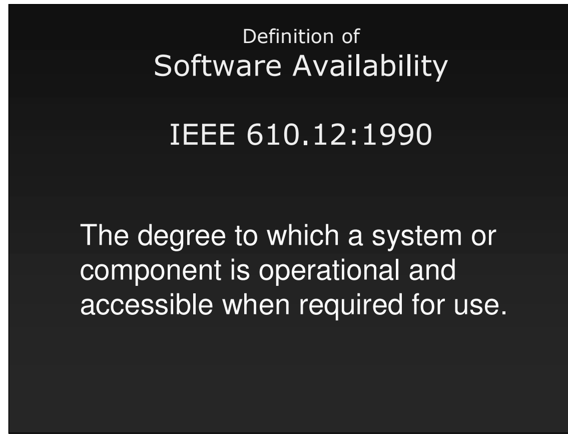
42. The use of certain design and coding standards is known to lead to more maintainable systems, with the inclusion of comments in the code probably being the most obvious example. The C code above (which does work and outputs "Choo Choo" to the screen) provides an example of 'how *not* to' format and comment code for ease of change.

43. Setting requirements about the maintenance and update of test case suites can also be considered as indirectly specifying the maintainability of a system. If test case suites are not available during the maintenance phase then it is obvious that extra time and effort will have to be spent performing regression testing if the quality of the software is to be maintained.

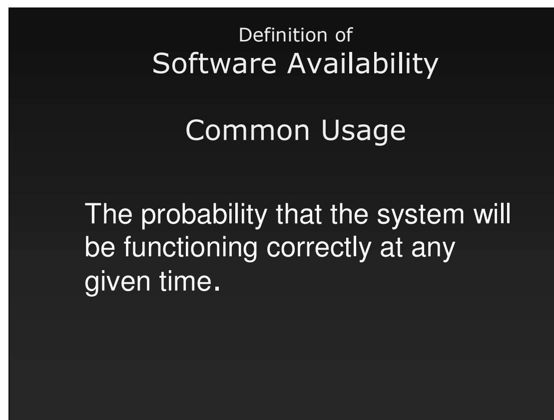
Maintainability Index

- The Maintainability Index [MI] aims to give a quantifiable method to measure the amount of effort required for maintenance
- 171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln (aveLOC) + 50 * sin (sqrt(2.4 * perCM))
  - aveV = average Halstead Volume V per module
  - aveV(g') = average extended cyclomatic complexity per module
  - aveLOC = the average count of lines of code (LOC) per module; and, optionally
  - perCM = average percent of lines of comments per module

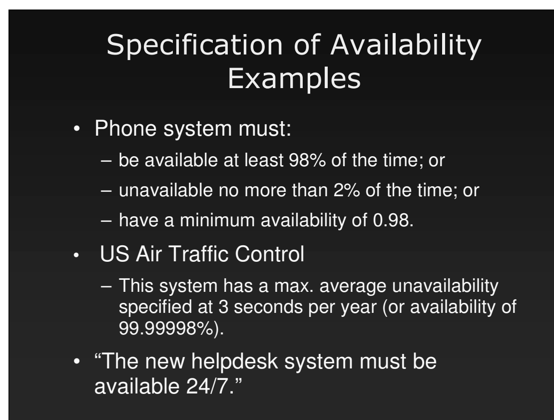
44. Another way of specifying (and testing for) software maintainability is the 'Maintainability Index', although many would consider this approach to be lacking credibility until both the theory behind the choice of values and more evidence on its effectiveness are made available (it was, however, produced by the SEI, the same organisation that produced CMM and CMMI). The 'Maintainability Index' uses a number of different metrics, including, Halstead's Volume and McCabe's Cyclomatic Complexity, both of which are outside the scope of this course. As can be seen above, it provides a relative value of maintainability for a complete software system that is supposed to relate directly to the amount of effort required to maintain it.



45. Having covered reliability and maintainability, the final component of dependability to consider is the availability of the software, the standard definition for which is shown above.



46. A clearer definition for most people is given above. Although the definition states availability as a probability, it is commonly stated in other ways.



47. As shown above with the phone system example, availability is often stated as a percentage, and may also be specified as unavailability, but can also be specified simply as a probability. The air traffic control example shows both the positive (availability) and negative (unavailability) approaches, while also showing how it may better be stated in terms of actual times. This real requirement also provides an example of an availability requirement that is practically impossible to achieve. The final example is absolutely impossible to achieve, as it

assumes the system will never fail.

Cost of Unavailability

Industry Sector	*Hourly Cost of Downtime (1996)*
Transportation	$90,000
Home Shopping	$113,000
Pay per view	$1,100,000
Credit Card Processing	$2,600,000
Brokerage	$6,500,000

48. The costs of not achieving high levels of availability are shown above for a number of US industry sectors in 1996 (presumably the costs are even higher today). The cost of a single hour's unavailability can be seen to be extremely high – figures such as these can be used to clearly demonstrate to project managers and customers the potential financial benefits of performing more testing – and so improving the system's availability.

Availability,
Reliability (MTTF) and
Maintainability (MTTR)

$$\textbf{Availability} = \frac{\textbf{MTTF}}{\textbf{MTTF} + \textbf{MTTR}} * 100\%$$

49. There is a close relationship between reliability, availability and maintainability. A (mythical) perfectly reliable system never fails and so is always available, and also requires no maintenance to fix faults (although maintenance to cope with changing user requirements is a different matter). When a system fails (and thus shows itself to be unreliable), maintenance is generally required to fix the fault that caused the failure. As we have already seen, the time required to fix a fault is indicated by the maintainability of the system. This relationship is shown in the formula above.

50. Given the formula, it can be seen that one means of testing the availability of software is to test its reliability and its maintainability and then combine the results to determine its availability.

51. As we have already seen, reliability testing is very closely related to functional testing (both are concerned with failures). Through the close relationship between availability and reliability, it can be seen that availability testing is also closely related to functional testing.

52. Availability is only slightly different from reliability, which you may remember was concerned

with being able to perform without failure. The differences are however great enough to require both attributes to be separately covered. For instance, consider a system managing a phone network. These systems can have surprisingly low levels of reliability, as long as they have very high levels of availability. The reason for this involves the consequences of failure. If a phone network 'drops' a call, then the caller is inconvenienced (and may think *they* dialled incorrectly), but is unlikely to suffer major harm. In fact, the caller will simply re-dial and as long as the system is already back up and running (available) then in most cases they will be relatively happy. Obviously this can't happen too regularly, but will be acceptable on an occasional basis. In terms of availability, then the caller wants the telephone system to be there practically all the time. To achieve this high availability when a failure occurs, the phone system must be able to recover very quickly to be able to connect the caller when they retry.

Definition of
Software Recoverability

ISO 9126

The capability of the software product to re-establish a specified level of performance and recover the data directly affected in case of failure.

53. This leads us to consider another software attribute – that of recoverability, which is defined above. In this definition it can be seen that time does not appear explicitly. However, in most cases we are worried about both whether the software can recover and, given that it can recover, how long it takes to do so.

Definition of
Software Recovery

IEEE 610.12:1990

The restoration of a system, program, database, or other system resource to a prior state following a failure or externally caused disaster.

54. A second term, closely related to recoverability is 'software recovery' defined above.

55. When specifying software recoverability many of the same difficulties apply as seen earlier with reliability specification. For instance, what is meant by a failure needs to be carefully defined - does the software have to be able to recover only from specific forms of hardware

failure, or also from software failures? Also, the operational conditions in which the software is to perform (its environment) need to be specifically stated as they can have a large impact on factors such as the software's speed of recovery. Finally, many systems do not recover to exactly the same level of operation as prior to the failure, but instead may recover to a degraded level, or may even 'fail-over' to a backup system until the primary system is able to take over again. In these cases the acceptable levels of recovered operation (and corresponding time periods) after failure need to be clearly stated.

Definition of
Recovery Testing

BS 7925-1:1998

Testing aimed at verifying the system's ability to recover from varying degrees of failure.

56. The testing of the ease with which software recovers from a failure is known as both 'recoverability testing' and 'recovery testing', defined above. This form of testing is normally aimed at the ability of the software to recover from both hardware and software failures, and is also normally concerned with automated recovery rather than that where human intervention is required.

57. Recovery testing normally takes two forms, both of which require the software to be run in an environment as close to the expected operational conditions as possible. The first is similar to that used for reliability growth testing, where pseudo random inputs based on an operational profile are used and failures occur 'naturally'. In the second faults are artificially 'introduced' into the system. With either approach when failures do occur the ability, or not, of the software to recover from the failure is recorded. In a typical recovery test a number of 'recoveries' are recorded and the recoverability is averaged out over the complete set.

Summary:
Software Dependability

- Reliability
- Maintainability
- Availability
- Recoverability

58. To summarise this section on Software Dependability, we have covered the quality characteristics of reliability, maintainability, availability and recoverability. Recoverability is not included in the ‘official’ definition of dependability, however, its close relationship to availability warranted its inclusion in this section.

Self-Assessment Questions

Q1. Which one of the following statements is correct?

(a) A failure occurs because a person made a mistake.
(b) A fault in the software leads to lower reliability.
(c) For every failure there is a corresponding fault.

Q2. Which one of the following sources of information is the least likely be used when creating an operational profile?

(a) Monitoring system use during defect testing.
(b) Interviewing customers on the expected users.
(c) Watching users with a prototype of the system.
(d) Determining the expected operational environment of the system.

Q3. Which one of the following values for probability of failure on demand (POFOD) would you expect to be specified for the primary protection system of a nuclear power station?

(a) POFOD = 0.0001.
(b) POFOD = 0.01.
(c) POFOD = 0.001.
(d) POFOD = 0.1.

Q4. Which one of the following measures of reliability would you expect to be used for a software system controlling the water levels in a reservoir?

(a) Mean time to failure (MTTF).
(b) Probability of failure on demand (POFOD).

Q5. Which one of the following quality characteristics does not appear in the IEC definition of dependability?

(a) Recoverability.
(b) Availability.
(c) Reliability.
(d) Maintainability.

Q6. Which one of the following is not a typical basis for specifying required values for software reliability?

(a) Developers' experience level.
(b) Possible system failures.

(c) Contractual agreement.
(d) Regulatory standard.

Q7. Which one of the following is not used as a means of achieving software reliability?

(a) Fault classification.
(b) Fault tolerance.
(c) Fault prevention.
(d) Fault removal.

Q8. Which one of the following quality characteristics is least likely to conflict with high reliability?

(a) Availability.
(b) Maintainability.
(c) Performance.

Q9. Which one of the following is least likely to be used to predict software reliability?

(a) Software Class Models.
(b) Software Process Models.
(c) Software Property Models.
(d) Software Fault Models.

Q10. Which one of the following is least likely to be used to determine whether indirect measures of software reliability have been achieved?

(a) Dynamic testing.
(b) Static analysis.
(c) Reviews.

Q11. Which one of the following is likely to provide the most accurate assessment of software reliability?

(a) Software Reliability Growth Models.
(b) Software Process Models.
(c) Software Property Models.
(d) Software Fault Models.

Q12. Which one of the following is not a major drawback of MTTR as a measure of software maintainability?

(a) It is an indirect measure of maintainability.
(b) We do not how the system will change in the future.
(c) The scale of changes varies dramatically.

Q13. Which one of the following is not considered to be an indirect measure of software maintainability?

(a) The number of faults in the system.
(b) Use of design and coding standards.
(c) Availability of regression test suites.
(d) The software design method used.

Q14. Availability can be defined in terms of reliability and maintainability. Which one of the following measures does not appear in the formula?

(a) POFOD.
(b) MTTR.
(c) MTTF.

Q15. Which one of the following statements is not correct?

(a) A perfectly reliable system is faultless.
(b) A perfectly reliable system never fails.
(c) A perfectly reliable system is always available.
(d) A high availability system may regularly fail.

Q16. Which one of the following is least likely to be specified when setting the recoverability requirements for software?

(a) The software integrity level of the application.
(b) The definition of failure for the application.
(c) The operational environment of the application.
(d) The acceptable level of recovery for the application.

Self-Assessment Answers

Note that the correct answer for all questions is currently set to (a).

Q1. Which one of the following statements is correct?

(a) A failure occurs because a person made a mistake.

Good – a mistake by a person leads to a fault in the software that, in turn, leads to a failure.

(b) A fault in the software leads to lower reliability.

No – some faults can be in the software but never 'activated' to cause a failure. Normally a special set of circumstances need to occur for a fault to cause a failure.

(c) For every failure there is a corresponding fault.

No – some faults can cause several failures and some failures can be caused by a combination of two or more faults.

Q2. Which one of the following sources of information is the least likely be used when creating an operational profile?

(a) Monitoring system use during defect testing.

Good – defect testing takes place rather late in the life cycle and also any profile based on defect testing is unlikely to be representative of operational use.

(b) Interviewing customers on the expected users.

No – this is one of the main ways of determining how the software is likely to be used.

(c) Watching users with a prototype of the system.

No – this is one of the main ways of determining how the software is likely to be used.

(d) Determining the expected operational environment of the system.

No – the operational environment of the system is a key part of the operational profile.

Q3. Which one of the following values for probability of failure on demand (POFOD) would you expect to be specified for the primary protection system of a nuclear power station?

(a) POFOD = 0.0001.

Good – this was the original requirement for the Sizewell B nuclear reactor primary protection system.

(b) POFOD = 0.01.

No – this corresponds to one failure in 100 – or, to put it another way, it will fail on

average on the 50th use.

(c) POFOD = 0.001.

No – although this is the value achieved for the Sizewell B nuclear reactor primary protection system, it was originally specified a factor of 10 higher.

(d) POFOD = 0.1.

No – this corresponds to one failure in 10 – or, to put it another way, it will fail on average on the 5th use.

Q4. Which one of the following measures of reliability would you expect to be used for a software system controlling the water levels in a reservoir?

(a) Mean time to failure (MTTF).

Good – this is a measure appropriate for use with a continuously running system, such as the reservoir monitoring system.

(b) Probability of failure on demand (POFOD).

No - this is a measure appropriate for use with a demand-driven system, while the reservoir monitoring system is a continuously running system.

Q5. Which one of the following quality characteristics does not appear in the IEC definition of dependability?

(a) Recoverability.

Good – although obviously closely related to availability, this term is not part of the IEC definition.

(b) Availability.

No – “availability performance” is this is the primary term used in the IEC definition.

(c) Reliability.

No – “reliability performance” is included in the IEC definition as an “influencing factor”.

(d) Maintainability.

No – “maintainability performance” and “maintainability support performance” are both included in the IEC definition as a “influencing factors”.

Q6. Which one of the following is not a typical basis for specifying required values for software reliability?

(a) Developers' experience level.

Good – the experience of the developers may influence the design chosen to achieve a level of reliability, but will not affect the requirement.

(b) Possible system failures.

No – working backwards from possible failures (and their impact) is one approach to deciding the required level of reliability.

(c) Contractual agreement.

No – many customers feel that they know the appropriate level of reliability (perhaps from past experience), but care must be taken if the system is safety-related that the contractual level is high enough to meet regulatory standards.

(d) Regulatory standard.

No – if the system is safety-related there will be a standard that defines the level of reliability required for the system (normally based on software integrity levels).

Q7. Which one of the following is not used as a means of achieving software reliability?

(a) Fault classification.

Good – knowing the types of fault does not help you achieve a level of reliability.

(b) Fault tolerance.

No – fault tolerance (building redundancy into the system) is one of the three main approaches to achieving software reliability.

(c) Fault prevention.

No – fault prevention (using good development techniques) is one of the three main approaches to achieving software reliability.

(d) Fault removal.

No – fault removal (testing and subsequent debugging) is one of the three main approaches to achieving software reliability.

Q8. Which one of the following quality characteristics is least likely to conflict with high reliability?

(a) Availability.

Good – availability is closely related to reliability and many, if not all, of the indirect measures of these two attributes are the same.

(b) Maintainability.

No – high reliability will mean redundancy will have to be built into the system, which

will add extra code and extra complexity, both of which militate against maintainability.

(c) Performance.

No - high reliability will mean redundancy will have to be built into the system, which will add extra code and so add extra processing time which militates against performance.

Q9. Which one of the following is least likely to be used to predict software reliability?

(a) Software Class Models.

Good – class models are part of a UML-based software development approach.

(b) Software Process Models.

No – the process used to develop the software is thought to influence the final reliability of the product.

(c) Software Property Models.

No – the properties of the software (complexity, etc.) are thought to influence the final reliability of the product.

(d) Software Fault Models.

No – the number of faults found during development are thought to have a correlation with the final reliability of the product.

Q10. Which one of the following is least likely to be used to determine whether indirect measures of software reliability have been achieved?

(a) Dynamic testing.

Good – dynamic testing is more appropriate for direct measurement of the software reliability.

(b) Static analysis.

No – static analysis is often used to measure attributes such as code structure, which can be used to help predict reliability.

(c) Reviews.

No – reviews of the design provide feedback on the quality of the design, which can be used to help predict reliability.

Q11. Which one of the following is likely to provide the most accurate assessment of software reliability?

(a) Software Reliability Growth Models.

Good – these models are the most mature method available for directly measuring the reliability of a system.

(b) Software Process Models.

No – although used, these models provide an indirect measure based on the assumption that good process leads to good product.

(c) Software Property Models.

No – although used, these models provide an indirect measure based on the assumption that software properties such as code complexity correlate with a level of reliability.

(d) Software Fault Models.

No – although used, these models provide an indirect measure based on the assumption that fault counts correlate with a level of reliability.

Q12. Which one of the following is not a major drawback of MTTR as a measure of software maintainability?

(a) It is an indirect measure of maintainability.

Good – MTTR (mean time to repair) is a <u>direct</u> measure of software maintainability.

(b) We do not how the system will change in the future.

No – not knowing how the system will change in the future is a major drawback to predicting how much effort (MTTR) is going to be required to change it.

(c) The scale of changes varies dramatically.

No – one change might mean a single line of code change, while another (which sounds very similar when described at the user level) might mean changing thousands of lines of code thus making the mean time to repair difficult to predict.

Q13. Which one of the following is not considered to be an indirect measure of software maintainability?

(a) The number of faults in the system.

Good – there is no obvious correlation between the number faults found in the code and how easy it is to modify the code.

(b) Use of design and coding standards.

No – design and code produced to good standards (which are generally aimed at maintainability) should be more maintainable.

(c) Availability of regression test suites.

No – if regression test suites are available it is easier to perform maintenance in the future as the tests can be reused.

(d) The software design method used.

No – it is generally agreed that some design methods (notably the object-oriented methods) lead to software that is easier to modify.

Q14. Availability can be defined in terms of reliability and maintainability. Which one of the following measures does not appear in the formula?

(a) POFOD.

Good – POFOD (probability of failure on demand) is not a time-based measure of reliability and so is not appropriate for determining availability, which is time-based.

(b) MTTR.

No – the formula is: $\textbf{Availability} = \frac{\textbf{MTTF}}{\textbf{MTTF} + \textbf{MTTR}} * 100\%$

(c) MTTF.

No – the formula is: $\textbf{Availability} = \frac{\textbf{MTTF}}{\textbf{MTTF} + \textbf{MTTR}} * 100\%$

Q15. Which one of the following statements is not correct?

(a) A perfectly reliable system is faultless.

Good – a system may have many faults, but if they are never ‘activated’ to cause a failure then the system will be perfectly reliable. Normally a special set of circumstances need to occur for a fault to cause a failure.

(b) A perfectly reliable system never fails.

No – reliability is based on failures, so one that never fails must be 100% reliable.

(c) A perfectly reliable system is always available.

No – a 100% reliable system never fails, and if it never fails it must always be available.

(d) A high availability system may regularly fail.

No – availability is based on the system functioning correctly at any one time, but as long as failures are recovered from quickly the overall amount of unavailable time will be low, thus still giving it high availability.

Q16. Which one of the following is least likely to be specified when setting the recoverability requirements for software?

(a) The software integrity level of the application.

Good – although the SIL might help decide the level of recoverability it is not needed to specify it.

(b) The definition of failure for the application.

No – unless it is agreed what is meant by a failure then the developer will not know in what circumstances they must build in the recovery.

(c) The operational environment of the application.

No – the operational environment of the application (e.g. the operating system, other applications interacting with our application) will have a major impact on its recoverability.

(d) The acceptable level of recovery for the application.

No – unless this is agreed the developers will not know whether the application has to be recovered to its original state or whether a degraded level of operation is acceptable.

References

http://www.de.ioccc.org/1986/marshall.c - Obfuscated C program that prints "choo choo". The obfuscated C site has lots of examples of impenetrable C programs, enjoy ☺

-- End of document --