

# SOFTWARE RELIABILITY TRAINING

## Proposed Class Schedule

<b>Session</b>	<b>Day 1</b>	<b>Day 2</b>	<b>Day 3</b>
1	Objectives Views of SWR	Failure causes, prevention & mitigation	Standards and texts
2	Characterization of Faults and Failures	Interpretation of test results, FRACAS	Failure prevention in OO software
3	Measures and Models of SWR	Interpretation of trends	Reliability of COTS and legacy software
4	Qualitative evaluation of SWR	Analysis techniques: FMEA & fault tree	SWR activities in the life cycle

### **1A. Objectives:**

The overall objective is to enable participants to plan and execute a software reliability or reliability improvement program, including

1. understanding the difference between software and hardware reliability and the role of software in system reliability
2. familiarity with the terminology and measures of SWR and applicable regulations and standards
3. ability to use test results for assessing and improving SWR
4. the difficulties encountered in demonstrating high reliability by test
5. use of analysis techniques for mission critical and real-time software
6. tailoring the plan for use with OO development and legacy software

### **1B. Product vs. Process**

1. There is much emphasis on improving software productivity by controlling the process. The process controls and feedback associated with them are also expected to improve software quality and thereby software reliability (the connection between quality and reliability is discussed later). The evidence for being able to achieve high reliability by relying primarily on process control is still lacking. One of the reasons may be that controls are centered on design and coding whereas serious faults are frequently introduced in the requirements phase and they are not detected by test.
2. Process controls must be integrated with the management style of each enterprise. It is difficult for an outsider to make meaningful contributions toward changing it.
3. For these reasons we concentrate here on achieving high reliability by analyzing the product and making changes to the product when deficiencies are found. However, these changes can be extended to the process by generalizing. We cover an example of this in Lecture 6

### **1C. Three views of software reliability**

1. The Purist (last seen in public ca. 1975 but may still hide in caves or abandoned structures) – software is a logical construct and as such is either correct or incorrect. Our contract specified correct software. Therefore we have a basis for rejecting incorrect software. Reliability is a numbers game and a waste of time.
2. Software Professional – the reason for the reliability problem on this project is that we weren't given enough time and budget. True, there were many failures initially, but look at the recent history of corrective action requests. In a couple of months we are going to be down to the noise level.

Corrective Maintenance Requests for XYZ System

Month	Number
January	18
February	20
March	13
April	9

3. System Engineer – the poor SWR is killing this project. You've got to do something!

Failure Report for XYZ System

Month	Hardware	Software	Skinware
January	40	24	21
February	32	35	22
March	18	46	15
April	22	60	12

Discussion: Can these views be reconciled?

### **2. Characterization of Faults and Failures**

1. Faults and failures – faults exist in a document (code, manual, ROM load sheet); failures are events, tied to execution of the program. A fault leads to a failure if the code segment is accessed with parameters that cause incorrect output.
2. Faults have a time of creation and a time of detection. The latter is usually known; the former is usually unknown but may be reconstructed. Faults can be classified as logical, arithmetical, procedural (indexing, etc), interface and violation of standards. Many faults are due to lack of understanding of system requirements and use. These classifications can aid in setting up training programs or to educate reviewers. The classifications are seldom mutually exclusive and exhaustive. The orthogonal defect classification (ODC) proposed by Chillarege attempts to overcome these problems.
3. Failures can be characterized by frequency of occurrence and by severity of effects (when they occur during test the severity may have to be assumed). Effects can be evaluated at the

software level (crash) and at the system level (inability to launch). From the combination of frequency and severity a “risk” (to the mission or project) can be assigned.

4. Faults are sometimes considered a software quality attribute, failures are always a reliability attribute. We will consider both under reliability.
5. There can be faults without failures, but never failures without faults (the faults may not be in the code)

### **3. Measures and Models of Software Reliability**

1. Failure rate measures and the related models have the form (No. of Failures)/(Time or Units of Use).
2. Numerator can be: all reported failures, all unique failures (deleting duplications), all repeatable failures, all failures above a given severity level, all failures that caused service interruptions above a threshold. Sometimes failures that are due to included legacy software are deleted from the count.
3. Denominator - Time can be: raw calendar time (per week, month, etc.), in-use calendar time (per hour of wall-clock usage), computer time (from log), execution time (from log or instrumentation). Units of Use can be: per transaction, per mission, per program execution, per instructions processed.
4. Due to the wide variety of numerator and denominator options it is very difficult to compare failure rate measures between projects and between developers. Agreements and standards are required.
5. Failure rate models can be: structure based, similar to a hardware reliability block diagram. Because of conditional transfers between blocks a Markov model is usually preferred to a block diagram. Other models consider the input domain (assigning failure probability to each type of input), and the application (assigning failure rates to guidance, communication, mission control, etc.). Fault depletion (reliability growth) models are the most frequently used ones and are discussed separately.
6. Under some simplifying assumptions the failure rate (for an execution-time or computer-time model) is proportional to the fault content. As faults are detected and fixed, the failure rate will then decline. But as the failure rate declines, fewer faults will be detected and the rate of improvement will decrease. The Goel-Okumoto, Musa, Schneidewind and Littlewood-Verall models will be discussed with examples from AIAA Recommended Practice for Software Reliability, R-013-1992. Automated forms of these models are available from SMERF. A combined Markov and fault depletion model will be demonstrated.
7. Fault density models have the form (No. of Faults)/(Units of Code)
8. The numerator of the fraction can be all faults in the code plus documentation errors, faults in executable code only, all faults after a given milestone (such as unit test), only faults verified by a review board, and combinations of these conditions
9. The denominator can be lines of source code (including or excluding comments), object code instructions, function points, number of modules or related measures.
10. The caution cited in 4. above applies here also.

#### **4. Qualitative Evaluation of Software Reliability**

1. Sometimes a numerical reliability requirement is imposed, such as a maximum failure rate of  $10^{-6}$  per flight hour. This is extremely difficult and expensive to demonstrate by test. Even for the best possible outcome of no failures, it will require about  $3 \times 10^6$  flight-equivalent hours of operation. Therefore we investigate qualitative techniques.
2. Evaluation of Space Shuttle Software, Deep Space Program telemetry, and other well-documented projects has shown that the causes of software failures change as test progresses and as the software matures. Initial failures may be due to faults in frequently accessed programs, then failures in simple exception handling predominate, and later failures under compound (multiple) exception conditions are detected. Examples of the progression in failure causes will be presented.
3. When a test produces only failures under compound exception conditions it is possible to parse the expected frequency of the conditions. Assume the failure occurs under simultaneous loss of electric power and a repeated NAK on a communication line. Experience indicates that the loss of power can be expected once per  $10^4$  hours of flight, and that the repeated NAK will occur once every  $10^3$  flight-hours. If the events can be shown to be independent, the compound failure can be assigned a probability of  $10^{-7}$  per flight-hour.
4. Techniques for (a) generating compound failure conditions and (b) evaluating them for reliability demonstration will be discussed.
5. The inverse relationship between severity and frequency can also be used for these demonstrations.

#### **5. Failure Causes, Prevention and Mitigation**

1. Most serious failures are due to deficiencies in requirements (missing, ambiguous, inconsistent, wrong). Because we know this, we have reviews of the requirements (at the system and software levels) and of products derived from the requirements.
2. The reviews catch some but not all of the deficiencies. Sometimes not all required skills are brought into the review cycle. Examples of required skills are shown in the following table.

### Example of Skills Required for Reviews

Phase	Cause	Sub-cause	Functional Specialist	System. Engineering	Design & Test	Software Team
Requirements	Mission not understood			x		
	Mode not specified (Exception handling)	Single condition	x	x		
		Multiple conditions		x		x
Design	Requirement not implemented				x	x
	Interactions not understood		x	x		
	Attribute of implementation not understood	Innovation	x			
		Legacy		x		
Implementation	Coding Error					x
	COTS/Middleware operation	Function not understood	x			x
		Interface not understood				
	Requirement not in test plan	Not considered relevant or testable				x
		Deliberately omitted		x		
	Test plan error	Ambiguous requirement		x		x
		Operation of UUT not understood			x	
	Test operation error				x	
	Test result incorrectly interpreted	Ambiguous requirement				x

3. Fault tolerance and fault containment techniques can overcome the effects of some faults as shown in the following table. Except where triple redundancy with voting is employed (extremely expensive and wasteful of computer performance) the fault (or failure) must be detected in order to activate the fault tolerance provisions. The detection can be at the software level (checksums, out-of-sequence operations, out-of-bounds address, etc.) or at the system level (acceleration or velocity error, loss of target acquisition, etc.)

## Capability and Cost of Fault Tolerance Techniques

Technique	Capability	Development Cost
Diverse element redundancy	Covers most faults, except due to common requirements	Very high
Similar element redundancy	Covers only faults due to timing and temporary data storage	Low
Standby with limited capability (lifeboat)	Same as diverse element redundancy but limited capability after switchover	Moderate
Checkpointing and restart	Covers only temporary effects	Low
Safe shutdown	Prevents unsafe conditions but shuts down further operation	Low

Discussion: For which applications would you use each of these techniques?

### 6. Interpretation of Test Results and FRACAS Reports

1. If no failures are reported in a test report this can be due to (a) very high quality software, or (b) inadequate testing. Similarly, the lack of reportable events in a Failure and Corrective Action System (FRACAS) Report can mean that the system performed very satisfactorily or that it did not see much use.
2. Each reported failure needs to be investigated at least at the levels indicated in the following table.

Investigative Step	Purpose	Examples of Action Taken
Immediate	Return system(s) to operating state	Work around, placard against some conditions of use
First level correction	Fix responsible condition	Code correction or patch
First cause assessment	Find reason for fault and why it was not detected	Review of requirements, design documents, test reports
Broad cause assessment	Find similar conditions in other software segments	Define conditions where this fault may exist, review software for these
Preventive Measures	Prevent this type of fault in future projects	Review of programming and test practices

Discussion: Which of the investigative steps is the responsibility of the failure review board?

3. Importance of FRACAS – It can be the best feedback on the effectiveness of quality and reliability measures but is frequently ineffective. Reasons: Failure reports are not filled out properly and not reviewed, considered a configuration management rather than quality and reliability tool, time lag between event and availability of reports.
4. Possible Improvements: Use of internet to make reports available earlier, compulsory fields, tools that take the drudgery out of report preparation

## **7. Interpretation of Trends**

1. Trends in Test Outcomes – attention to test exposure: number of runs in a given time interval, number of off-nominal conditions investigated, number of test instrumentation errors or difficulties.
2. Trends in test outcomes – number of anomalies reported: review for multiple failures due to a single fault, classify for severity of failures, probability of encountering the conditions that led to the failure.
3. Evaluation of trends in outcomes after normalization for severity and probability of causative conditions – can they be fitted to one of the reliability growth models in SMERF?
4. Comparison between predicted and actual trends can help us find weak spots in test procedures. Examples from Y2K tests.
5. Other trends: Cluster Analysis can identify common causes or an underlying cause. Look for clusters in time (field data), clusters by software component, clusters by environment or associated events.

Discussion: What makes for trends in test outcomes? Do test procedures provide the uniform stress level that is assumed in the growth models?

## **8. Analysis Techniques – FMEA and Fault Tree**

1. FMEA is an essential tool for hardware reliability programs. The format of FMEA is defined in MIL-STD-1629. It has seen only limited use in software because
  - a. software is not made up of parts
  - b. it is much more difficult to define failure modes in software
  - c. effects propagation is more difficult to evaluate because of branchingComputer-based tools for hardware FMEA are usually not suited for software.
2. Object oriented programs permit a more organized application of FMEA to software
  - a. objects are similar to parts
  - b. methods constitute failure modes
  - c. effects propagation can be evaluated with development toolsUML-based tools can automate some steps of software FMEA.
3. Combined hardware/software FMEA has potential benefits but is still experimental. Many software failures occur when software is called on to diagnose or recover from hardware failures. This area needs further evaluation
4. Fault tree analysis is a top-down method (most hardware FMEA proceeds bottom-up). A “top event” (hazard or mission impairment) is identified, and all paths that can produce that event are analyzed.
5. Probabilities are assigned to the “leaves” of the tree (the lowest level) and are then propagated upward with AND and OR gates. This yields the probability of the top event.
6. Some computer based tools are available for organizing FTA.

Discussion: In what situations will you want to use software FMEA? Software FTA? Is there synergy between them?

## **9. Standards and Texts**

### 1. General Use Standards and Handbooks

ANSI/AIAA R-013	MIL-STD-882C
RTCA DO-178B	IEEE-Std-730
MIL-HDBK 338, Section 9	IEEE-Guide-982.1 and .2

### 2. Texts

Musa, Iannino, Okumoto “Software Reliability”  
Musa “Software Reliability Engineering”  
Nancy Leveson “Safeware”  
Kopetz “Real-Time Systems”

### 3. NASA Publications

NHB 1700.1 Preliminary Hazards Analysis  
NASA STD 8719.13A Software Safety  
NASA GB 1740.13 Guidebook for Safety Critical Software

Content and applicability of these will be discussed/

## **10. Failure Prevention in Object Oriented Software**

1. UML based development tools capture and organize requirements and avoid many causes of failure introduced at the start of software development
2. Inheritance offers reliability benefits *if failure experience on the parent object is utilized.*
3. Analysis of the methods in use and properties utilized at the time of a test failure can speed up the analysis and facilitate broader cause assessment (see Interpretation of Test Results)
4. Message Sequence Charts can be analyzed for timing related reliability problems.

Discussion: What has been the reliability experience with OO development? How has test case generation been affected?

## **11. Reliability of COTS and Legacy Software**

1. Common problems of COTS and legacy software: Differences between the intended application and the application of previous use can negate the supposed benefits of using “tried and true” programs. The same is true for changes at the interfaces to support the new environment.
2. Common activities for COTS and legacy software: Compare usage profile in intended application with that of the previous one. Significant differences can invalidate the reliability

experience. Careful review of any test failures for underlying causes that may not have been corrected, particularly in variable names and data declarations.

3. COTS specific problems: Reliability data may be difficult to get, and if they are provided they are usually not in a form that can be propagated to the new software environment. COTS features that are not utilized should be deleted or disabled to prevent unwanted interference with intended operations (execution of code can occupy resources even if no output is produced).
4. Legacy software specific problems: Incompatibility in support software (compilers, models, test case generators, manuals) can create reliability problems

Discussion: What reliability criteria should be used for decisions to develop vs. COTS or legacy?

## 12. Software Reliability Activities in the Life Cycle

SWR participation in reviews and other milestone events is usually governed by the program plan. The following table lists activities that are particularly significant.

Phase	Activity	Examples
Requirements	Generate SWR objectives from system reliability requirements	Max. failure rate for a specified scenario at a given milestone
		Max. failure ratio in acceptance test
	Plan for verification	Estimate number of test cases
	Assess realism	Compare to prior achieved reliability
Prelim. Design	Apportion SWR goals to segment	Analyze participation of each segment in the specified scenario
	Select SWR tools	Review development tools for support of SWR
		Procure additional tools and training
	Plan for achievement of objectives	Establish failure rates to be achieved by each segment at prior milestones
Integration of SWR activities with system reliability	Common or compatible tools and models, compatible reliability measures	
Design & Code	Populate models	Reliability estimates based on segment size and utilization
	Review test plan for SWR issues	Type and number of test cases, compatibility with scenario for achieving SWR goal
	Review unit tests	Type of failures and errors encountered, look for common causes
	Start analysis activities	Establish framework for FMEA and FTA
Test	Participate in FRB	Evaluate data for SWR significance
		Evaluate FRACAS format for broad cause assessment
	Review test preparation	Determine that test problems do not obstruct data collection for SWR
	Data analysis	Assess satisfaction of SWR objectives
	Trend analysis	Achievement of reliability growth objectives
	Analysis of failure causes	Broad cause assessment and formulation of corrective action.
Operation	Participation in FRB	Activities as in test
	Analysis of failure data	Determine consistency with data from prior systems and from test, formulate corrective action

Discussion: Have we learned enough to plan SWR activities for our next project?