An overview of the IEEE 1633 Recommended Practices for Software Reliability

- Introduction and motivation to product this guidance
- Solutions provided by the guidance
- Quantitative and qualitative reliability measures for making a software release decision

Ann Marie Neufelder
ann.neufelder@missionreadysoftware.com

IEEE 1633 Working Group members

- Robert Stoddard - SEI
- Lance Fiondella - UMass
- Peter Lakey - Consultant
- Robert Binder – retired
- Michael Siok – Lockheed Martin
- Ming Li - NRC
- Ying Shi - NASA
- Nematollah Bidokhti - thinkDFR
- Thierry Wandji – US Navy
- Michael Grottke - FAU
- Andy Long - OSD
- George Stark - IBM
- Allen Nikora - NASA
- Bakul Banerjee – retired IEEE
- Debra Greenhalgh Lubas – US Navy
- Mark Sims – US Army
- Rajesh Murthy - Consultant
- Willie Fitzpatrick – US Army
- Mark Ofori-kyei – General Dynamics
- Sonya Davis – General Dynamics
- Burdette Joyner – Northrup Grumman
- Marty Shooman –retired NASA
- Andrew Mack
- Loren Garroway – Northrup Grumman
- Kevin Mattos– US Navy
- Kevin Frye - US Navy
- Claire Jones - Boeing
- Robert Raygan - OSD
- Mary Ann DeCicco – General Dynamics
- Shane Smith - OSD
- Franklin Marotta – US Army
- David Bernreuther – US Army
- Martin Wayne – US Army
- Nathan Herbert – US Army
- Richard E Gibbs III - Boeing
- Harry White - Harmonic
- Jacob Axman – US Navy
- Ahlia T. Kitwana - Harris
- Yuan Wei
- Darwin Heiser – General Dynamics
- Brian McQuillan – General Dynamics
- Kishor Trivedi – Duke University
- Debra Haehn – Philips Healthcare

Chair: Ann Marie Neufelder, Mission Ready Software

Vice Chair: Lance Fiondella - UMass

Secretary: Rachel Neufelder, Mission Ready Software

IEEE Standards Association Chair: Louis Gullo, Northrop Grumman

Martha Wetherholt of NASA was Vice Chair until her passing in 2020. She was instrumental in delivering the 2016 edition.

Introduction and Motivation

Reliable software engineering…

- Process assessments such as the SEI CMMi assessment have not provided value added for improving reliability or safety of software
- 30 years of data shows [1]
  - No improved software reliability beyond level 3
  - Organizations with CMMi level 3+ can and do produce failed software programs
  - A good process is necessary *but not sufficient*
  - An organization can have a great process and still have
    - People who do not understand the product or industry doing development and test
    - Low level of rigor in testing
    - Requirements that are traceable but poorly written
    - Design that is traceable but poorly written
    - Test procedures that are traceable but poorly written and have low coverage
    - Overlooked failure modes
    - Too many unknown defects in the software
    - Too many known open defects that aren't assessed appropriately
    - Too many workarounds in the software that burden the end users and/or cause loss of availability

Introduction and Motivation

Reliable software engineering…

Has been an engineering disciple for > 50 years.

Fundamental prerequisite for virtually all modern systems

Plenty of theory generated over last several decades, but…*Practical guidance on how to apply these models has lagged significantly*

Diverse set of stakeholders requires pragmatic guidance and tools to apply software reliability models to assess real software or firmware projects during each stage of the software development lifecycle

Fundamental roadblocks addressed by IEEE 1633

- Reliability engineers don't understand software
- Software engineers don't understand reliability
- Both may have challenges acquiring data needed for the analyses

Solutions provided by IEEE 1633

Actionable step by step procedures for assessing reliable software

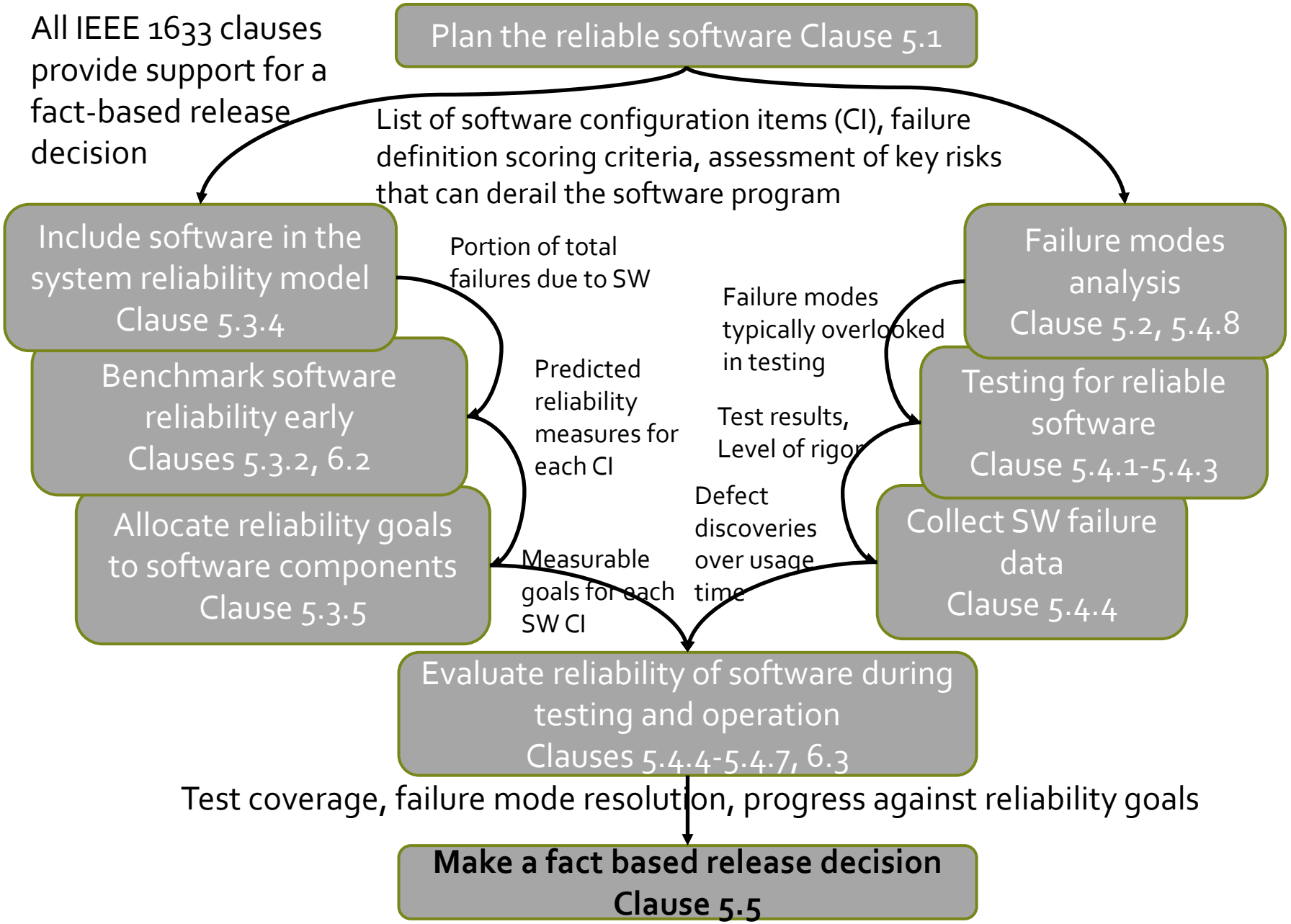
During **any phase** of software or firmware development

With **any software lifecycle model** for any industry or application type.

- Reliable software is demonstrated by both qualitative and quantitative evidence
- Fact based decisions for releasing software based on qualitative and quantitative aspects
- Quantitative measures are the demonstration that software is reliable
- Qualitative measures provide confidence that quantitative measures are accurate
  - *The easiest way to record software fewer defects is to test less or test with less rigor*

Qualitative	Qualitative
Estimated portion of system failures that will be due to software	Level of rigor in testing including • Test Like You Operate • Fault injection testing • Peak loading and endurance testing • Boundary and zero value testing • Code coverage • Go-No go testing • Requirements coverage
Rate of defect discovery in testing	
Fix rate versus open rate	
Estimated residual defects and potential for pileup	
Severity level/effect of unresolved defects	Failure mode identification against the "Common Defect Enumeration" or known set of software failure modes and/or fault tree analysis
Defect density benchmark	
Coverage metrics	Defect root cause analysis

All IEEE 1633 clauses provide support for a fact-based release decision



Current status of IEEE 1633

- Unanimously approved by IEEE Standards Association in first ballot of May 24, 2016. Released on January 18, 2017.
- Working group is currently making updates for
  - How reliable software tasks are executed in DevSecOps
  - Common Defect Enumeration recently published on the Defense Acquisition University R&M Community of Practice website
    - https://www.dau.edu/cop/rm-engineering/_layouts/15/WopiFrame.aspx?sourcedoc=/cop/rm-engineering/DAU%20Sponsored%20Documents/Reliable%20Software%20SOW%20Appendix%20B%20-%20CDE.xlsx&action=default
  - Level of rigor in testing

- Quantitative
- Estimated portion of system failures that will be due to software
- Rate of defect discovery in testing
- Fix rate versus open rate
- Estimated residual defects and potential for pileup
- Severity level/effect of unresolved defects
- Defect density benchmark
- Coverage metrics

Quantitative

- Exactly “1” quantitative measure should not be used as a release decision maker
  - Ex: Defects per source line of code or failures per hour should not be used without other metrics
  - No one metric tells the whole story
  - It’s too easy for one metric to be a self-fulfilling prophecy.

- Past history method is very accurate if the past history is recent and is calibrated for changes in technology.
- As a rule, software grows 10-12% per year. So, historical data should be calibrated to assume that the software portion is growing 10-12% per year.
- There is virtually no chance that software will decrease in size over time. Hence past history is a useful lower bound.
- Real example: An engineering company produced a system in 2015. Of all of the deployed failures, 25% were due to software. In 2017 they were deploying a similar system.
- Since historical data was 2 years old, 25% is adjusted by 10-12 % per year.
- So, the prediction is between 30.25% and 31.36%.
- When the equipment was deployed in 2019 - the actual portion of failures due to software was 33%. Much more accurate than the 5% estimated by subject matter experts.

Estimated portion of system failures due to software

Method	Description
Past history	Compute relative portion of SW versus HW failures from a past similar system
R&D $	Compute relative portion of R&D $ dedicated to software development
Achievable failure rates	Use prediction models to determine failure rate for HW, SW. The predicted values for each determine their allocation.

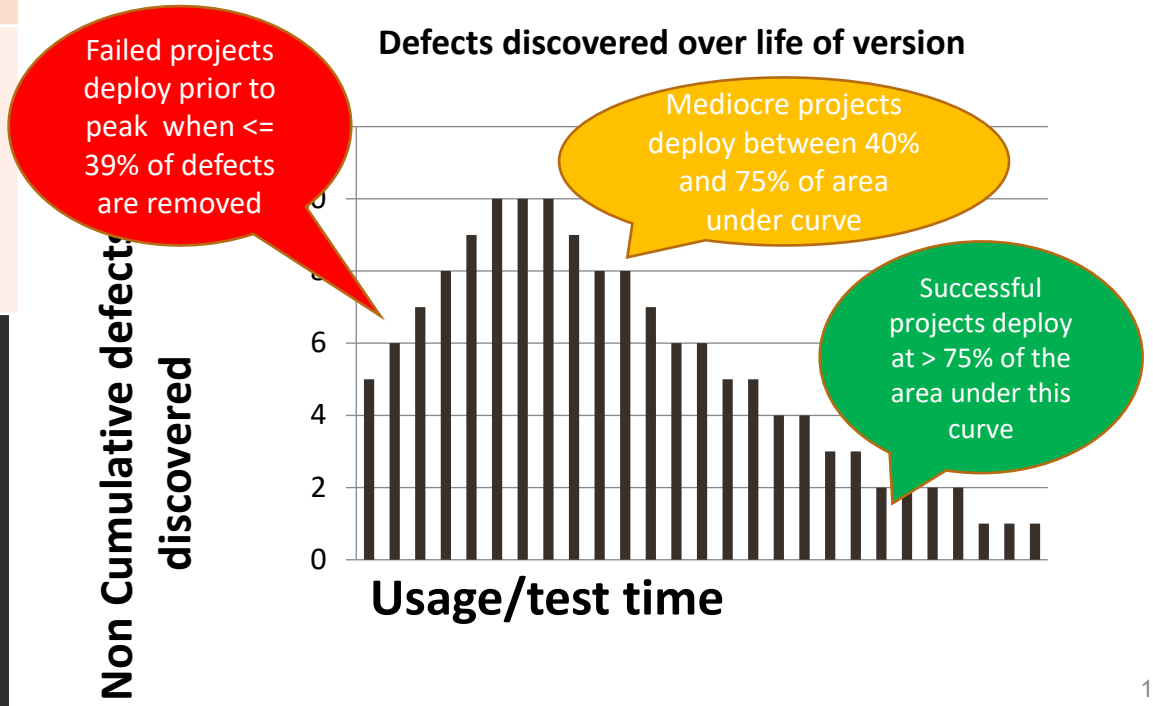
Value added: Engineering has no basis for assuming that software doesn't fail or that the reliability = 1

Rate of defect discovery in testing

Metric	World Class	Mediocre	Distressed
Fault rate trend	Steadily decreasing	Peaking or recently peaked	Increasing
Percentage of defects identified prior to deployment versus post deployment	>=75%	40-74%	<=39%

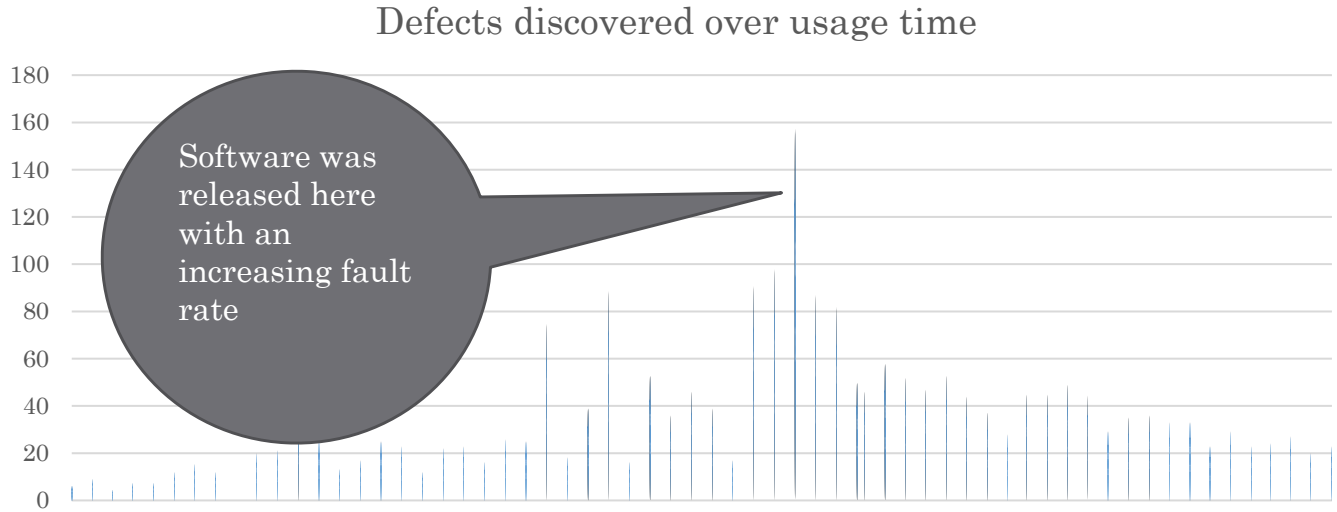
Value added: Ensures software isn't deployed half baked

- **Software fault rate increases, peaks and then decreases prior to maturity**
- Maturity level at deployment separates the world class from the distressed
  - Increasing fault rate– the customers will see it as a failed product in 100% of all cases
  - Fault rate barely decreasing- customers will be unhappy with it
  - Fault rate is steadily decreasing – customers won't notice the SW *which is ultimate indicator of success*
- With agile or incremental development there are multiple peaks until the final burn down of defects
- **We cover how to track fault rate during testing in the IEEE 1633 clause 5.4.4**



Lessons learned from a real software intensive program in which defect discovery rate was not tracked

This is the defect rate from a distressed software intensive program



The organization released the software to operational deployment before the fault rate peaked.

That's because no one was trending the fault rate.

More than 800 software failures were discovered by customer after deployment.

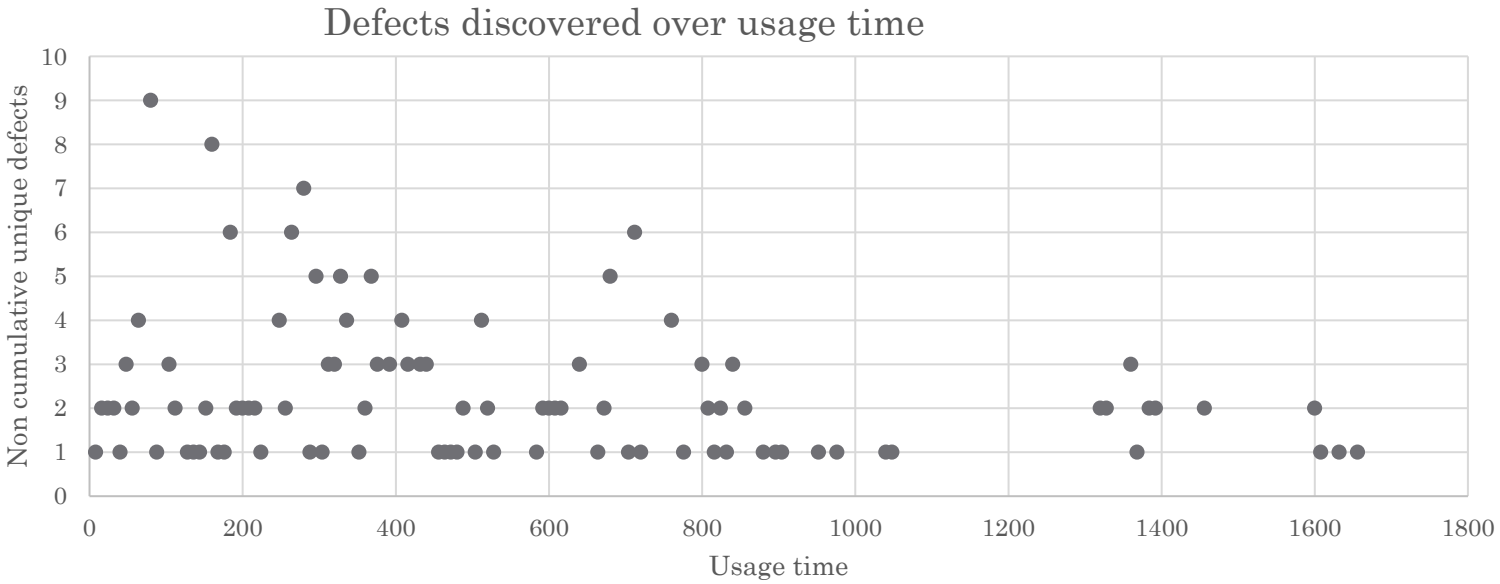
Upon deployment, the actual system reliability was **8 %** of the required reliability objective **because of the software failures**.

If SWRG models had been used prior to deployment, the service would not have accepted the software as is since the RAM goal had not been met.

Lessons learned from a real software intensive program in which defect discovery rate was tracked

This is the fault rate from a real software intensive program

- The fault rate is clearly trending downwards
- By the end of the trend, approximately 80% of defects had been discovered (The IEEE 1633 shows how to calculate this)
- There was still work to be done with regards to defect removal but the software is stable.
- The SWRG model provides confidence that the overall RAM objective ***can*** be met and the work required to meet it



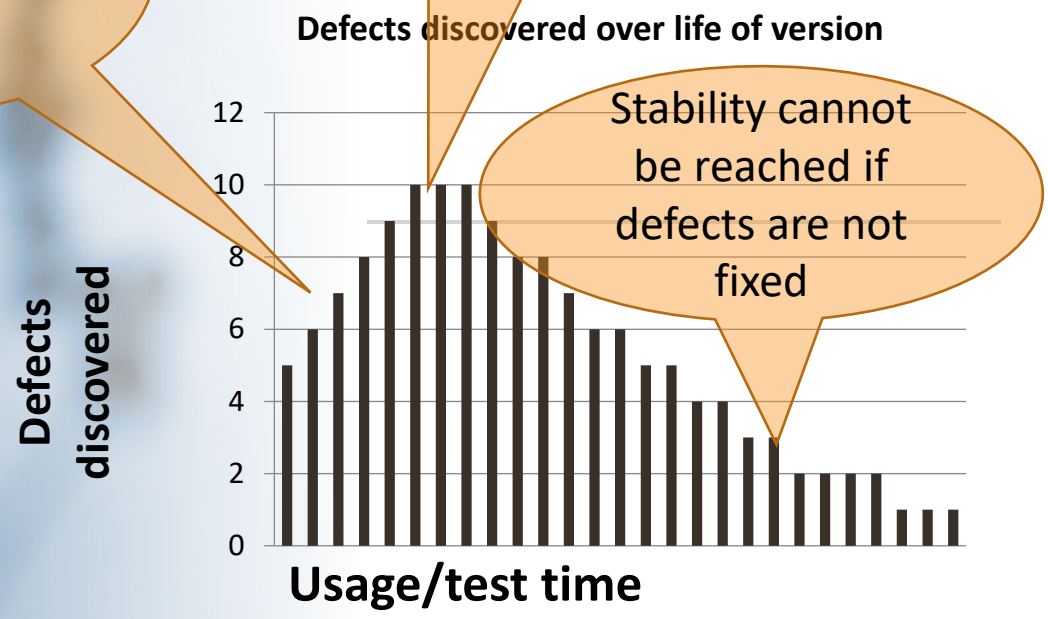
Fix rate versus open rate

If the discovered defects aren't removed - the defect discovery trend won't improve very far beyond the peak

Peak is reached when blocking defects are either fixed or avoidable

Increase is caused by blocking defects

Stability cannot be reached if defects are not fixed

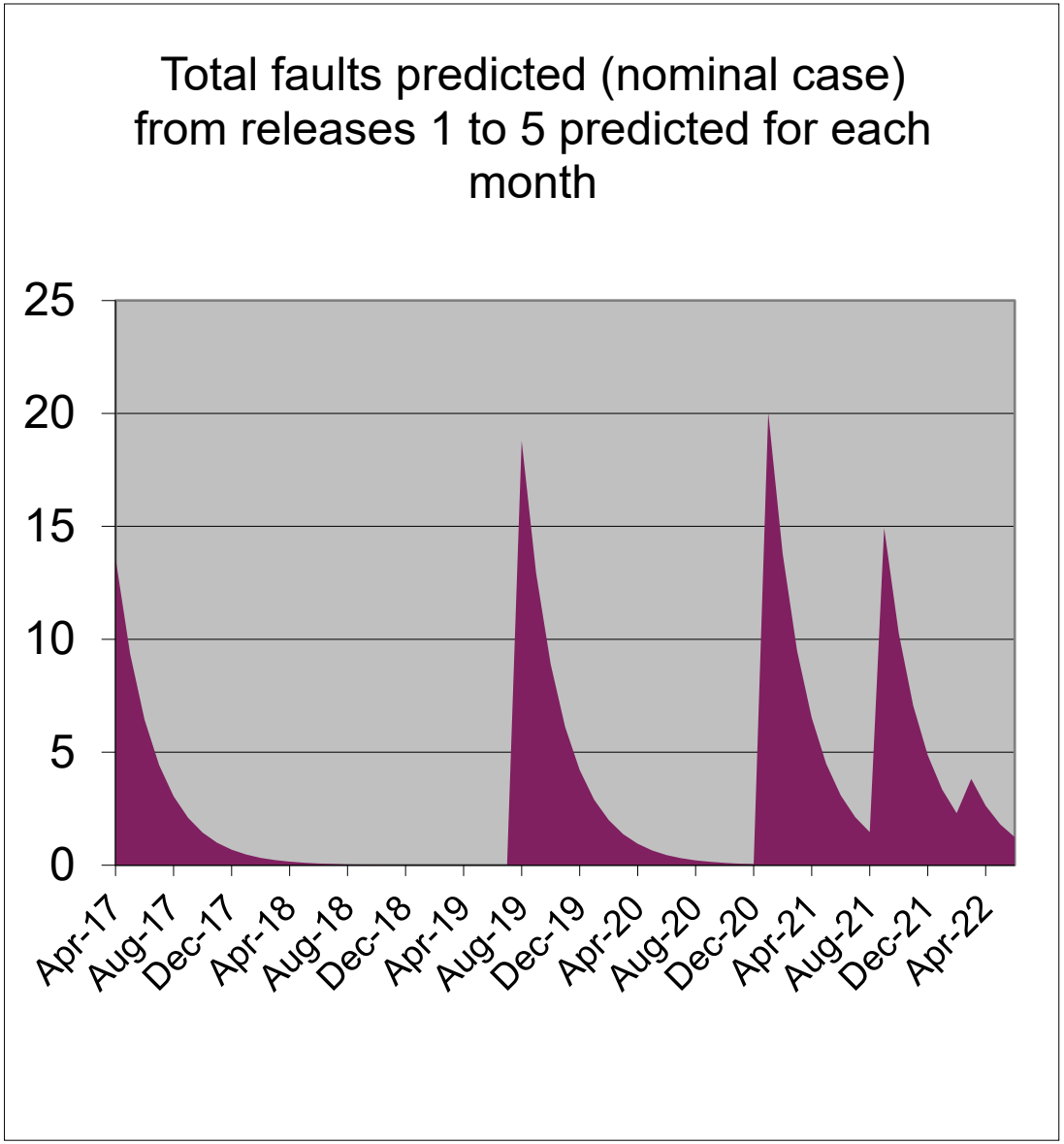


Value added: A stable defect discovery profile won't happen if the defects aren't removed

Estimated residual defects and potential for pileup

- Releases are too far apart initially and too close together in last 3 releases
- SRE predictions allowed for leveling of features before the code is even written

Value added: An early sprint or release might be stable; but at planned cadence eventually future sprints or releases won't be



Severity/effect of unresolved defects

Value added: Software isn't like hardware. Software can have the following failures that typically don't happen with hardware.

It's possible to have a decreasing defect trend and defects that aren't piling up but still have unreliable software because there are open defects that the user can't experience

1. A single defect causes multiple failures
2. The same failure happens at every installed site *at the same time*
3. A collection of defects "with workarounds" collectively cause the user to have unacceptable downtime
4. One "reset" could happen so often that the system is unusable
5. The worst software failures often happen when the software is executing
6. Systematic software failures need to be counted as a failure every time they occur until the underlying failure mode is proven to be removed or mitigated

Defect density benchmarking statistics

Cluster	Outcome	Defect metrics			Late deliveries (as per SW estimates)	
		Average defects per 1000 source lines of code	% defects removed prior to release	Fault rate	Prob (late)	How much project is late by as % of schedule
3%	World Class	.0269	>75%	Steadily decreasing	40	12
10%	Successful	.0644			20	25
25%	Above average	.111	40-75%	Recently peaked or recently decreasing	17	25
50%	Average	.239			34	37
75%	Below average	.647			85	125
90%	Impaired	1.119	<40%	Increasing or peaking	67	67
97%	Distressed	2.402			83	75

Value added: The defect profile can be predicted before testing even starts. Sometimes one bad development practice can derail the program. You don't want to wait until testing or operation to find that out.

- Tables like this are derived from actual field data [1][2]
- Organizations with lowest deployed defect density were also late less often and by a smaller amount
- SRE for any given project can be benchmarked by answering a simple survey

Type of factor	Number /% of characteristics in this category	Examples of characteristics in this category
Product	50 – (10%)	Size, complexity, whether the design is object oriented, whether the requirements are consistent, code that is old and fragile, etc.
Product risks	12 – (2%)	Risks imposed by end users, government regulations, customers, product maturity, etc.
People	38 – (7%)	Turnover, geographical location, amount of noise in work area, number of years of experience in the applicable industry, number of software people, ratio of software developers to testers, etc.
Process	121 – (23%)	Procedures, compliance, exit criteria, standards, etc.
Technique	302 – (58%)	The specific methods, approaches and tools that are used to develop the software. Example: Using a SFMEA to help identify the exceptions that should be designed and coded.

Static analysis tools measure these

These are often overlooked

SEI CMMi and ASPICE assess this

These are often overlooked

Factors that have been mathematically proven to be related to software reliability [1][2]

USAF Rome Laboratories developed first prediction model in 1987[2]. It was based on these factors.

A few more have been developed since then.

Facts don't lie. All predictive models agree that how the software is developed is a good predictor for its ultimate reliability.

Category	Examples
Decomposition	• Code a little, test a little philosophy. • Release development/test time < 18 months long and preferably <12 months. • Each developer has a schedule that is granular to day or week.
Visualization with pictures and tables	A picture is worth 1000 words. Specifications with diagrams/pictures/tables are associated with fewer defects than text.
Requirements focus	Developing requirements that aren't missing crucially important details
Testing focus/rigor	Explicitly testing the requirements, design, stresses, lines of code, operational profile
Unit testing focus	Unit testing by every software engineer is mandatary and as per a defined template. Branch coverage tools and metrics.
Defect reduction techniques	Software fault trees, software FMEA, etc.
Design focus	Designing states, sequences, timing, logic, algorithms, error handling **before coding**
Regular monitoring of the software engineers	Monitoring software progress daily or weekly, identifying risks early, etc.
Planning ahead	Planning the scope, personnel, equipment, risks before they become problematic, planning the timing of the tasks

Techniques that have been proven to effect software reliability that are often overlooked

Coverage metrics

Software Metrics	Definition
Requirements Traceability	Degree to which the requirements have been met by the architecture, code and test cases
Structural coverage	Degree to which the lines of code, paths, and data have been tested

Value added: The estimated fault trend and remaining defects are only as accurate as the amount of code and requirements that have been covered.

- These metrics put the defect profile observed in testing into perspective
- Ex: If only half of the requirements are covered or half of the code has been executed in testing then the reliability growth progress is optimistic by at least 50%. (i.e. total defects are at least twice what have been observed so far)

Qualitative

Level of rigor in testing including
- Test Like You Operate
- Fault injection testing
- Peak loading and endurance testing
- Boundary and zero value testing
- Code coverage
- Go-No go testing
- Requirements coverage

Failure mode identification against the “Common Defect Enumeration” or known set of software failure modes and/or fault tree analysis

Defect root cause analysis

Qualitative

- An organization can have excellent quantitative measures but still have unknown failures that cause serious problems for end users
- This is because every development activity is designed for success.
- Engineering calls these failures “edge” cases
- But really these failure modes were detectable all along. They just chose not to look for them or design to them.

Value added: Reliable software is achievable only when software is designed to avoid failure.

Type of test	Inputs	Description
Black box testing		
Operational profile testing	The Operational Profile (OP)	Also known as "Test Like You Operate"
Requirements based testing	The software requirements	Exercises the SUT to provide assurance that it satisfies its requirements as specified.
Model based testing	Test models may be derived from requirements and design documentation.	Exercises state transitions, fault states, dead states, prohibited states
Stress case testing	Expected longest mission time and max concurrent users	Peak loading, endurance, zero value testing, boundary values, go no-go testing
Timing and performance	Timing and scheduling diagrams, performance requirements, the effects of software on the system design	Exercises the SUT to evaluate compliance with requirements for real-time deadlines, resource utilization.
Failure modes – See next slide	Software FMEA, defect root cause analysis, fault trees	Exercises the conditions that are associated with the identified failure modes. This is the only test that verifies that software works properly in a degraded environment.

Develop a reliability test suite to maximize the level of rigor

Contrary to popular belief testing only the requirements is rarely sufficient.

Value added: These tests often identify defects that are the most severe and most expensive to fix once deployed

Software failure mode	Description
Faulty state management	Inadvertent state transitions, dead states, state transitions are incorrect, etc.
Faulty sequencing	Operations execute in the wrong order
Faulty timing	Operations start too early or too late or take too long. The right event happens in the right order but at the wrong time
Faulty data	The data is the wrong units of measure, scale, resolution, type, size, stale, corrupt, missing.
Faulty functionality	The system does the wrong thing or does the right thing incorrectly
Faulty error detection	The software fails to detect faults in the hardware, communication, computations, power, external components or devices, computations, etc.
Faulty processing	Software accuracy or memory degrades over mission, software can't handle peak loading or maximum users

Analyze failure modes and effects

- Problem: Engineers will often consider failure modes that are so obvious that they are guaranteed to be found in testing – or they will consider failure modes that aren't fixable in the software.
- There are over 400 failure mode/root causes that are at least relevant to all software systems.
- The failure modes are tagged to actual failures from mission and safety critical systems since 1962

Value added: Virtually 100% [3] of the software failure modes that have cause problems in operation are due to the above categories which are often overlooked.

History repeats itself.

Root causes are predictable but only if someone thinks about them.

- **Faulty error handling** – Apollo 11 lunar landing, ARIANE5, Quantas flight 72, Solar Heliospheric Observatory spacecraft, Denver Airport, NASA Spirit Rover (too many files on drive not detected)
- **Faulty data definition** - Ariane5 explosion 16/64 bit mismatch, Mars Climate Orbiter Metric/English mismatch, Mars Global Surveyor, 1985 SDIO mismatch, TITANIV wrong constant defined
- **Fault logic**– AT&T Mid Atlantic outage in 1991
- **Timing** - SCUD missile attack Patriot missile system, 2003 Northeast blackout
  - Race condition - Therac 25
- **Peak load conditions** - Affordable Health Care site launch, Iowa Primary
- **Faulty usability**
  - **Too easy for humans to make mistakes** – AFATDS friendly fire, PANAMA city over-radiation
  - **Insufficient positive feedback of safety and mission critical commands** –

The above illustrates that **history keeps repeating itself** because people assume root causes from other industries/applications are somehow not applicable.

Lesson to be learned – the root causes are applicable to any industry/product. It's the hazards/effects that result from the root causes that are unique.

Software failure modes effects analysis are highly effective **but only if the below 17 mistakes are avoided**

Organizational mistakes

- None of the software FMEA analysts have a background in software
- The analysis is not constructed by a cross functional team
- Conducting the SFMEA too late (most of these failure modes are too expensive to fix once the code is written)
- Conducting the SFMEA without the proper software deliverables such as the SRS, SDD, IRS, etc.
- Failing to track the failure modes and/or make any corrective actions to the requirements, design, code, use case, users manual as a result of the SFMEA
- Failing to tailor the software FMEA to the highest risk areas and most relevant failure modes

Faulty Assumptions

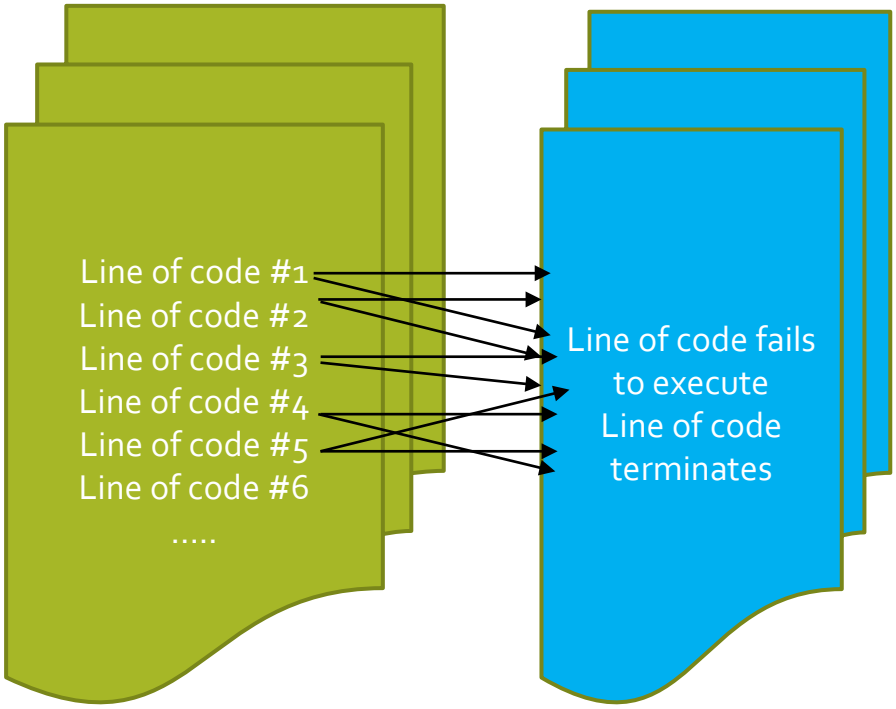
- Assumption that all failures originate in a single line of code or specification
- Assumption that software works
- Assumption that software specifications are correct and complete
- Assumption that all failure modes will be found and fixed in testing
- Assumption that all failure modes are impossible or negligible in severity

FMEA Execution mistakes

- Focusing on total failure of the software - failing to consider small things that lead to big things going wrong
- Black box versus functional approach – analyze what the software does and not what it is
- Ignoring the 6 dimensions that lead to software failures - the system, the users who use the system, the battlefield environment, and the mission
- Conducting the SFMEA at too high (system requirements) or too low (lines of code) a level or architecture
- Mixing functional failure modes with process failure modes (i.e. fault timing means the software design not the software schedule)
- Incorrectly assigning a failure rate or likelihood

Value added: The IEEE 1633 explains out how to apply the FMEA so that the 17 common mistakes are minimized. The recommended practice can be applied to any industry FMEA standard for framework.

FAULTY ASSUMPTION THAT ALL FAILURE MODES ORIGINATE IN A SINGLE LINE OF CODE



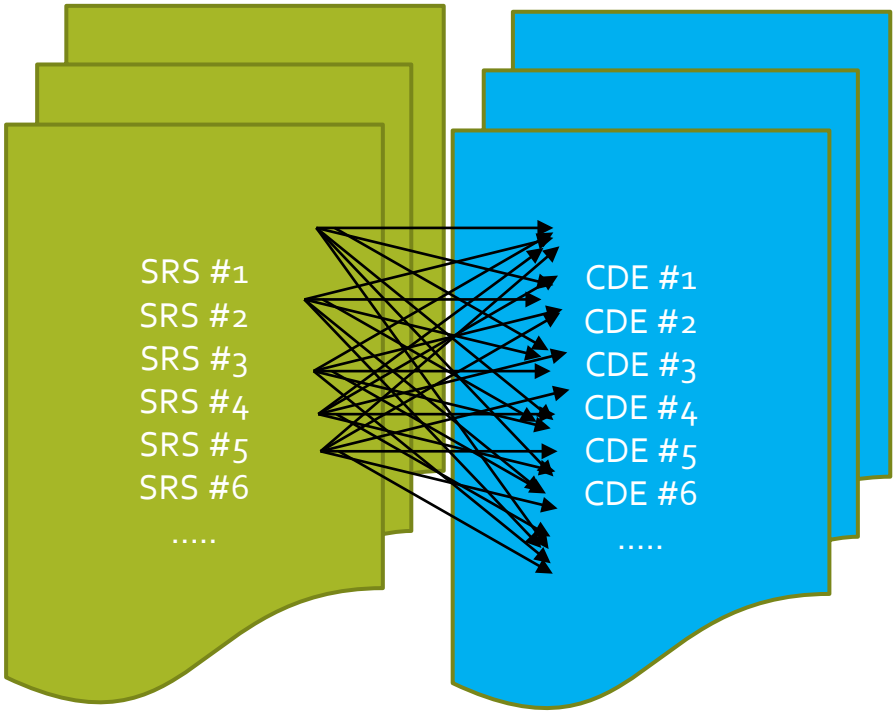
The analysts work through each line of code one at a time and analyze against statement each CDE one at a time.

This is ineffective because:

1. Very few failures are due to a **single** line of code [3]
2. When a failure is due to a single line of code it is usually due to mistakes like these
   - Line of code executes the wrong command (i.e. has a compilable typo)
   - Line of code manipulates the wrong data
   - Line of code uses isn't written properly but still compiles

3. Lines of code typically don't fail to execute unless there is a defect in another line of code
4. If a line of code terminates execution it is often because there is missing fault handling or by faulty design

FAULTY ASSUMPTION THAT ALL FAILURE MODES ORIGINATE IN A SINGLE SPECIFICATION



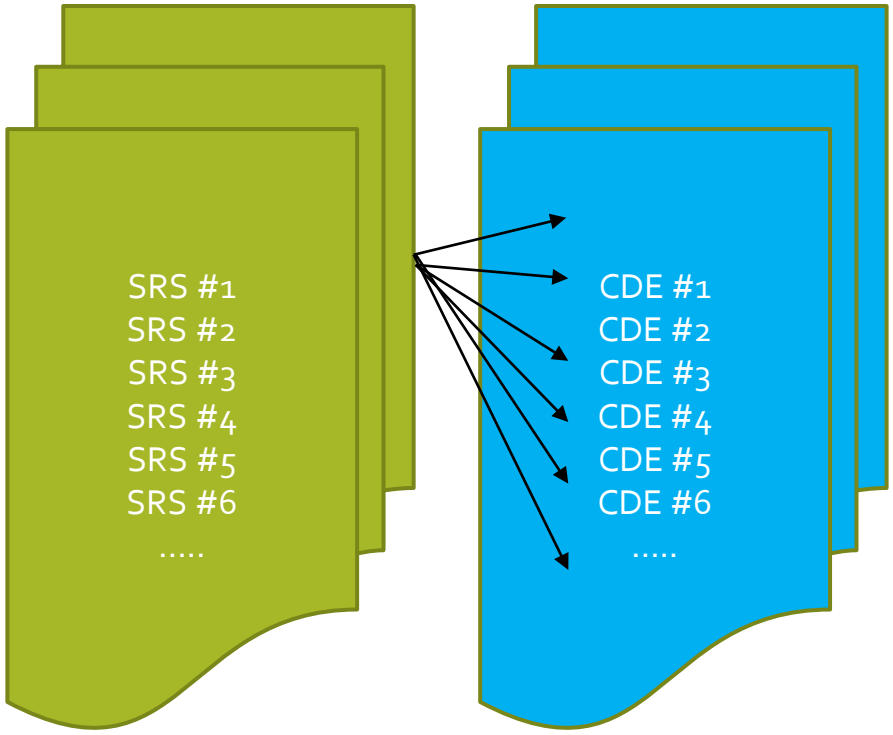
The analysts work through each SRS statement one at a time and analyze against statement each failure mode one at a time.

This is ineffective because:

1. Majority of operational defects aren't caused by a single faulty statement [3]
2. Many of the common defect enumerations don't apply at the statement level – they apply to a collection of statements
3. Primary failure mode at an individual specification is magic numbers (i.e. timing or accuracy requirements.)

INCOSE requirements analyzers are effective at identifying requirements statements that are ambiguous or untestable.

MANY FAILURES ARE DUE TO A COLLECTION OF SOFTWARE SPECIFICATIONS, MISSING SPECIFICATIONS AND MULTIPLE LINES OF CODE



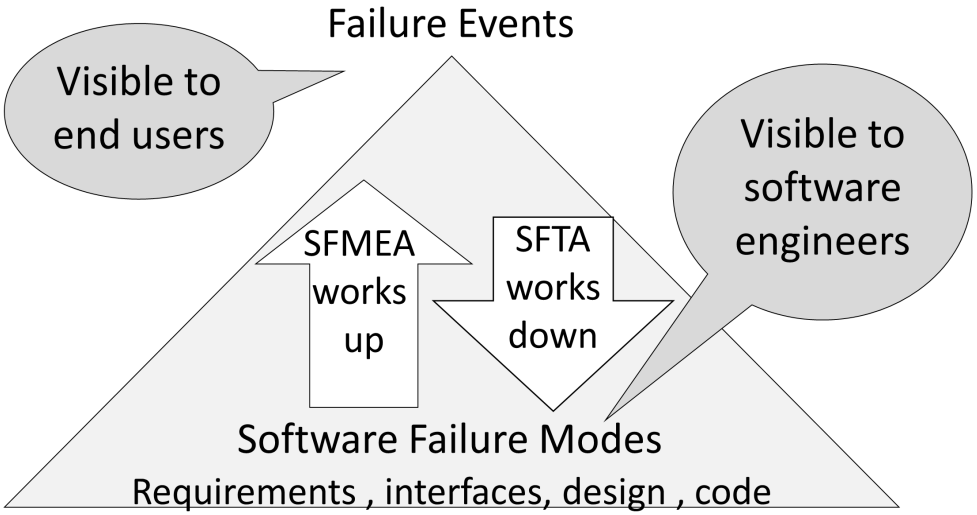
Analyze the collection of software requirements against the set of CDEs

1. Prune the CDEs to remove things you don't have in the software
2. Analyze the specifications and design as a whole package against the relevant CDEs

Value added: The Common Defect Enumeration lays out the failure modes that have caused the most failures in operation. The CDE can be used in any FMEA framework or industry standard.

Include software in a system fault tree

- A "software" fault tree should be part of an overall system fault tree to ensure that interactions with hardware are considered
- The fault tree can feed the software FMEA
  - The hazards are tagged to the top level effects in the SFMEA
  - If the SFMEA is effective it will cover every hazard
- The software FMEA can feed the FTA
  - It may/will identify hazards that weren't considered in the FTA, PHA, FHA



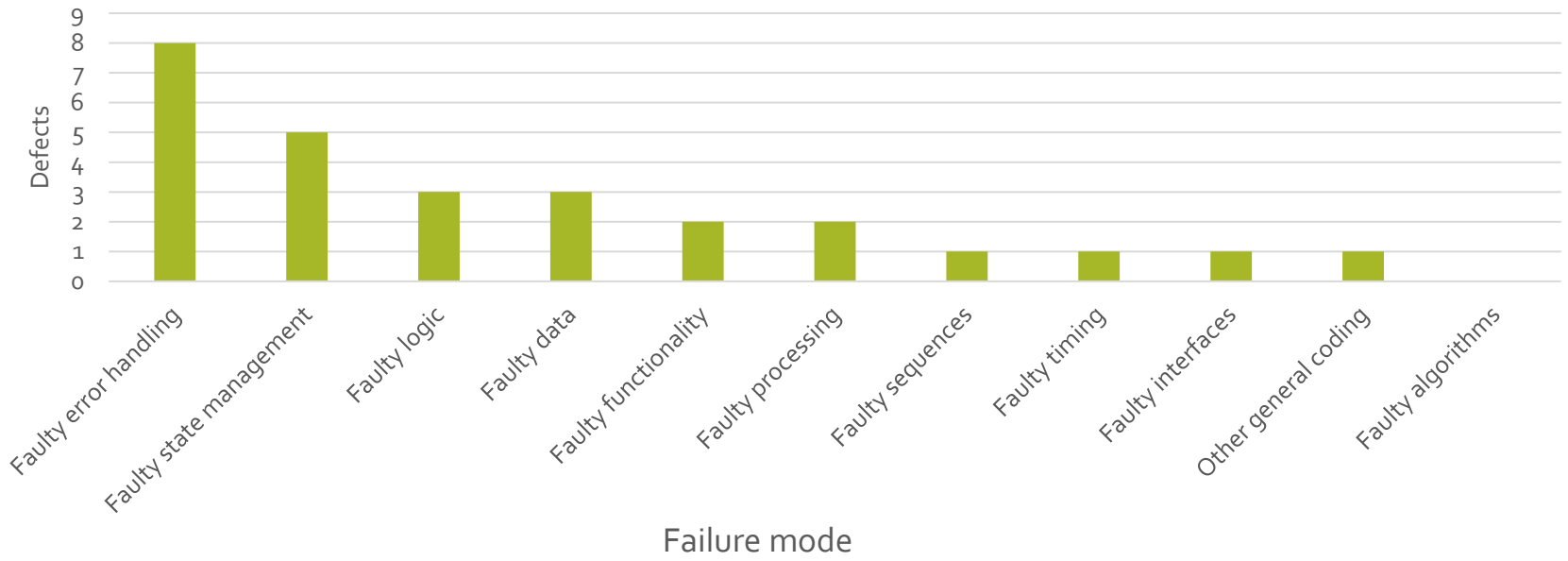
Value added: A fault tree can jump start the software FMEA to ensure that the most likely failure modes tagged to the most serious hazards are considered.

Software defect root cause analysis

- The defect RCA can be employed all by itself or prior to a software FMEA
- Defect RCA ensures that fault injection testing, design reviews, specification reviews, and code reviews, focus on the most relevant root causes for the application under development
- Defect RCA has 3 viewpoints
  - Defects by originating artifact
    - Contrary to popular belief most defects found in operation are not "coding" defects. They are specification and design defects that led to coding defects.
  - Defects by failure mode
    - Faulty timing, sequencing, state management, error handling, functionality, processing, logic, interfaces, etc.
  - Defects by root cause
    - See the Common Defect Enumeration [2]

Value added: Failure modes that have happened in the recent past are the most likely to happen again. That's because software engineers usually fix one instance of a defect but don't fix related systematic instances.

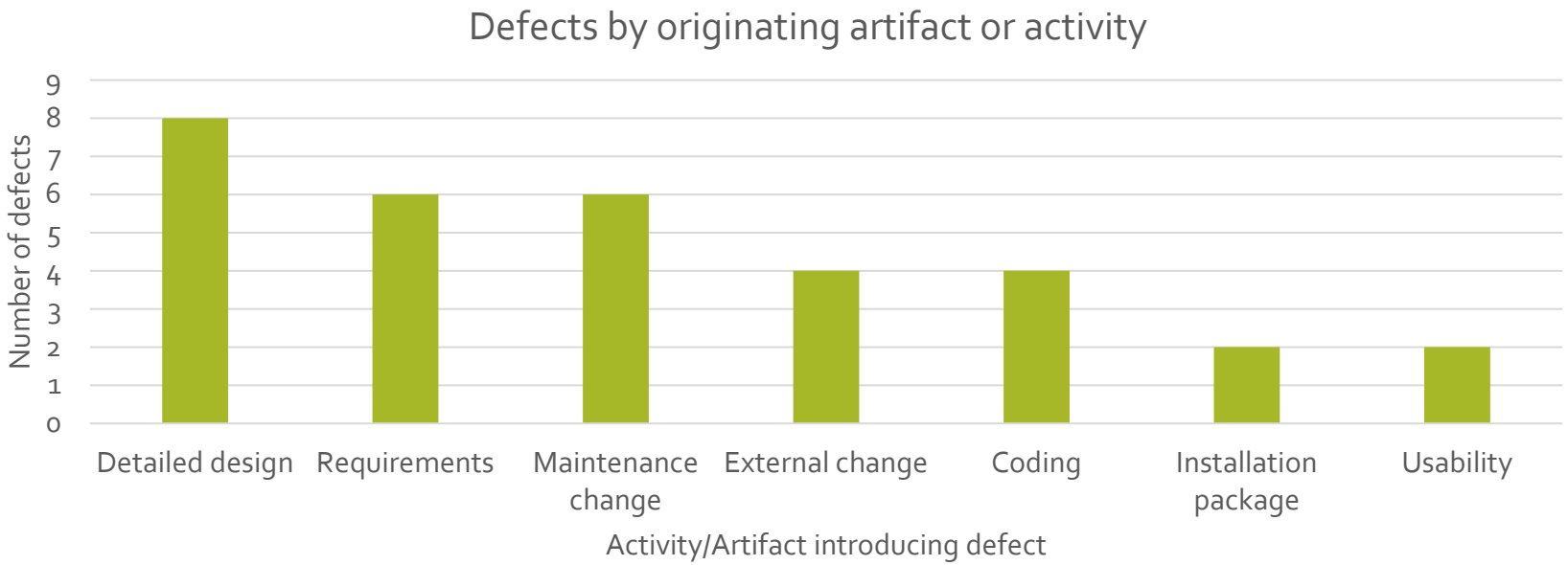
Software defect root cause analysis



The most common failure mode is directly related to the weakest link in the development activity. Examples:

- If the software engineers fail to consider that software must detect failures in hardware there will be more faulty error handling failure modes
- If the software engineers fail to do state diagraming prior to coding and the system is stateful; state management defects are more likely
- If the software engineers fail to do timing design and timing is important for the application; timing defects are more likely

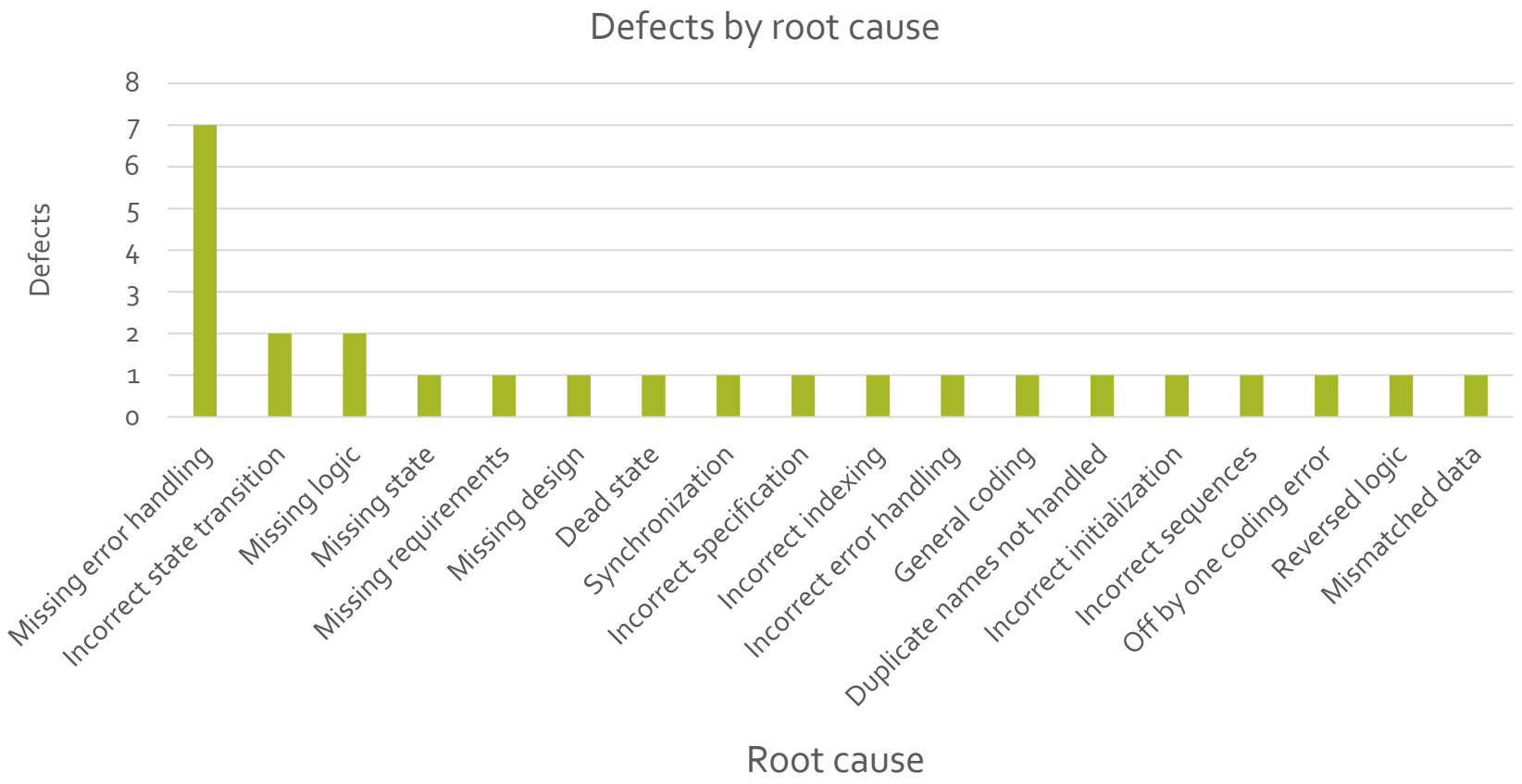
Software defect root cause analysis



Defects are introduced because of either bad requirements, bad design, bad coding practices or bad change control.

- Requirements defect – The "whats" are incorrect, ambiguous or incomplete.
- Design defect – The "whats" are correct but the "hows" are not. Logic, state, timing, exception handling are all design related.
- Coding defect- The "whats" and "hows" are correct but the software engineer did not implement one or more lines of code properly.

Software defect root cause analysis



Each of the failure modes has multiple root causes as per the Common Defect Enumeration [2]

Summary

IEEE 1633 2016 includes improved guidance over 2008 edition

Offers increased value to this audience

- Reliability engineers
- Software quality engineers
- Software and engineering managers
- Acquisitions
- Regulatory

IEEE 1633 2016 puts forth recommended practices to apply qualitative software failure modes analyses and qualitative models

Improve product and ensure software or firmware delivered with required reliability

IEEE 1633 2023 will make following improvements

Incorporates

- Common defect enumeration
- Tailoring for DevSecOps
- Updated models
- Refined guidance

References

[1] The Cold Hard Truth About Reliable Software, Version 6i, AM Neufelder, 2019.

[2] Rome Laboratory TR-92-52, “Software Reliability Measurement and Test Integration Techniques”, J Mccall, W Randell, J Dunham, L. Lauterback, 1992.

[3] The Common Defect Enumeration, AM Neufelder, Copyright Mission Ready Software, 2021.

https://www.dau.edu/cop/rm-engineering/_layouts/15/WopiFrame.aspx?sourcedoc=/cop/rm-engineering/DAU%20Sponsored%20Documents/Reliable%20Software%20SOW%20Appendix%20B%20-%20CDE.xlsx&action=default

[4] Effective Application of Software Failure Modes Effects Analysis, AM Neufelder, published by Quanterion Solutions, Inc., 2014.