Ένωση Πληροφορικών Ελλάδας

Αξιοπιστία Λογισμικού
Μαθήματα από τις Διαστημικές Αποστολές

Χάρης Γεωργίου (MSc, PhD)

https://www.epe.org.gr https://www.hiu.gr

Ένωση Πληροφορικών Ελλάδας

Στόχοι:

- Πρώτος "καθολικός" φορέας εκπροσώπησης πτυχιούχων Πληροφορικής.
- Αρμόδιος φορέας εκπροσώπησης επαγγελματιών Πληροφορικής.
- Αρμόδιος επιστημονικός "συμβουλευτικός" φορέας για το Δημόσιο.
- Αρωγός της Εθνικής Ψηφιακής Στρατηγικής & Παιδείας της χώρας.

Τομείς παρέμβασης

Ποιοι είναι οι κύριοι τομείς παρεμβάσεων της ΕΠΕ;

1. Εθνική Ψηφιακή Στρατηγική & Οικονομία
2. Εργασιακά (ΤΠΕ), Δημόσιος & ιδιωτικός τομέας
3. Παιδεία (Α′, Β′, Γ′)
4. Έρευνα & Τεχνολογία
5. Έργα & υπηρεσίες ΤΠΕ
6. Ασφάλεια συστημάτων & δεδομένων
7. Ανοικτά συστήματα & πρότυπα
8. Χρήση ΕΛ/ΛΑΚ
9. Πνευματικά δικαιώματα
10. Κώδικας Δεοντολογίας (ΤΠΕ)
11. Κοινωνική μέριμνα (ICT4D)

Harris Georgiou (MSc, PhD) – https://github.com/xgeorgio/info

- R&D: Associate post-doc researcher and lecturer with the University Athens (NKUA) and University of Piraeus (UniPi)
- Consultant in Medical Imaging, Machine Learning, Data Analytics, Signal Processing, Process Optimization, Dynamic Systems, Complexity & Emergent A.I., Game Theory
- HRTA member since 2009, LEAR / scientific advisor
- HRTA field operator (USAR, scuba diver)
- Wilderness first aid, paediatric (child/infant)
- Humanitarian aid & disaster relief in Ghana, Lesvos, Piraeus
- Support of unaccomp. minors, teacher in community schools
- Streetwork training, psychological first aid & victim support
- 2+4 books, 170+ scientific papers/articles (and 5 marathons)

Επισκόπηση – Πηγές

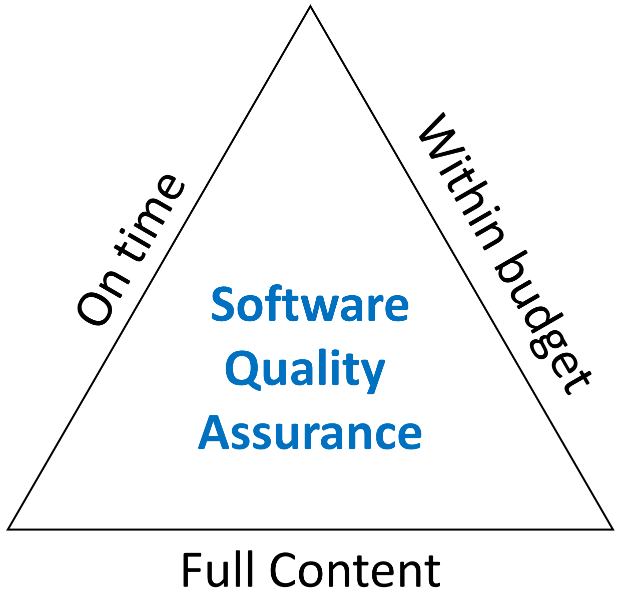
- Περιεχόμενα:
  - Τι είναι η «Αξιοπιστία Λογισμικού» και η Διασφάλιση Ποιότητας Λογισμικού (SQA).
  - Γιατί οι μεθοδολογίες διασφάλισης ποιότητας λογισμικού είναι απαραίτητες.
  - Μέθοδοι και μετρικές στις διάφορες φάσεις ανάπτυξης λογισμικού.
  - Μερικά παραδείγματα από τον πραγματικό κόσμο – Διαστημικές αποστολές.

- Πηγές:
  - ISO/IEC 9126 Software Quality Model (1993), ISO/IEC 25010 Software Quality Model (2011).
  - G. Schulmeyer, J. McManus, “Software Quality Handbook”, Prentice Hall (1998).
  - IEEE Std 730 (2002), IEEE Standard for Software Quality Assurance Plans, IEEE Computer Society / Software Engineering Standards Committee.
  - Rosenberg, L.H.; Gallo, A.M., Jr., “Software quality assurance engineering at NASA”, Proc. IEEE Aerospace Conf. 2002, Vol.5.

Software Quality Assurance (SQA): What is it?

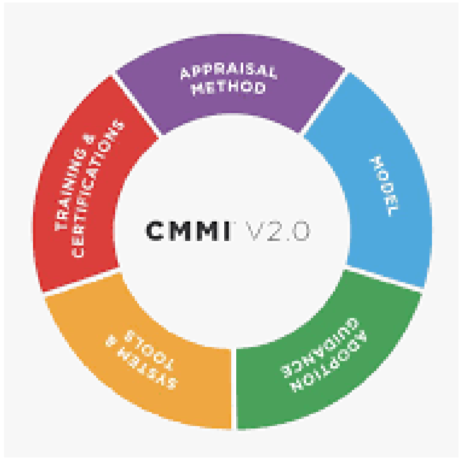
- **What is “quality”?**
- **IEEE Glossary**: “Degree to which a system, component, or process meets (1) specified <u>requirements</u>, and (2) customer or user <u>needs or expectations.</u>”
- **ISO**: “The totality of <u>features and characteristics</u> of a product or service that bear on its ability to satisfy <u>specified or implied needs.</u>”
- “Requirements” = needs / “Specifications” = design framework
- “Expectation management”: Some may not be fully implemented due to technological, budget or time constraints.

SQA: The three constraints



SQA: How do we achieve it?

- Work within a disciplined, methodological approach
- "Engineering" is predictable and repeatable
- Adopt a formal Quality Management System (QMS)
  - Process documentation
  - Best Practices & Guidelines
  - Formal design tools (user stories, UML, ...)
  - Formal systems (ISO 9001, CMMI, ...)
- Continuous Improvement
  - Audit / review processes
  - Software Metrics
  - Software Maintenance



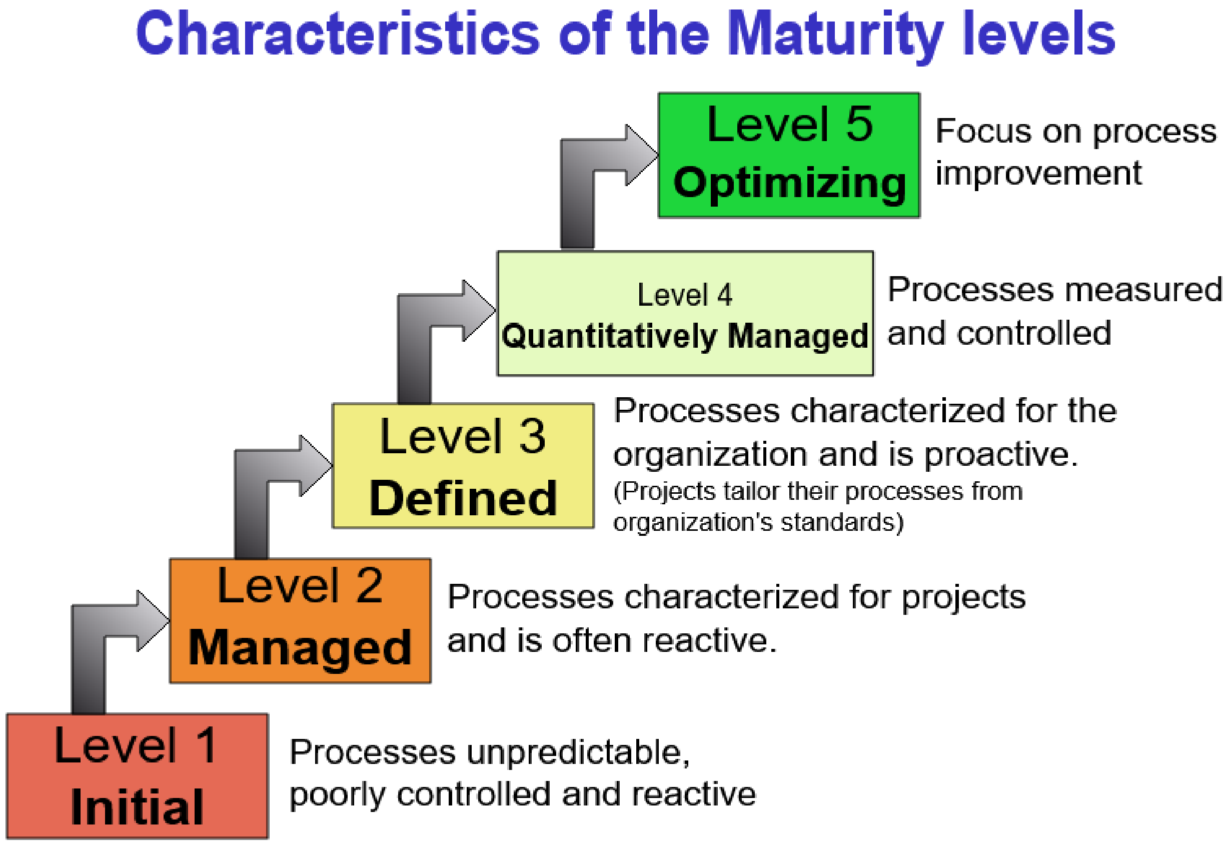
Quality Management Systems

- QMS define the “proper” way to do SE work
  - usually focus on the “what” rather than the “how”
  - adequately flexible and adaptable (see: ISO)
  - applicable in a wide range of contexts (not only SE)

- QMS = Procedures + Documentation + Guidelines
  - Procedures: defined mostly at top-level
  - Documentation: mandatory for audit & traceability
  - Guidelines: mostly recommendations (“should have”)

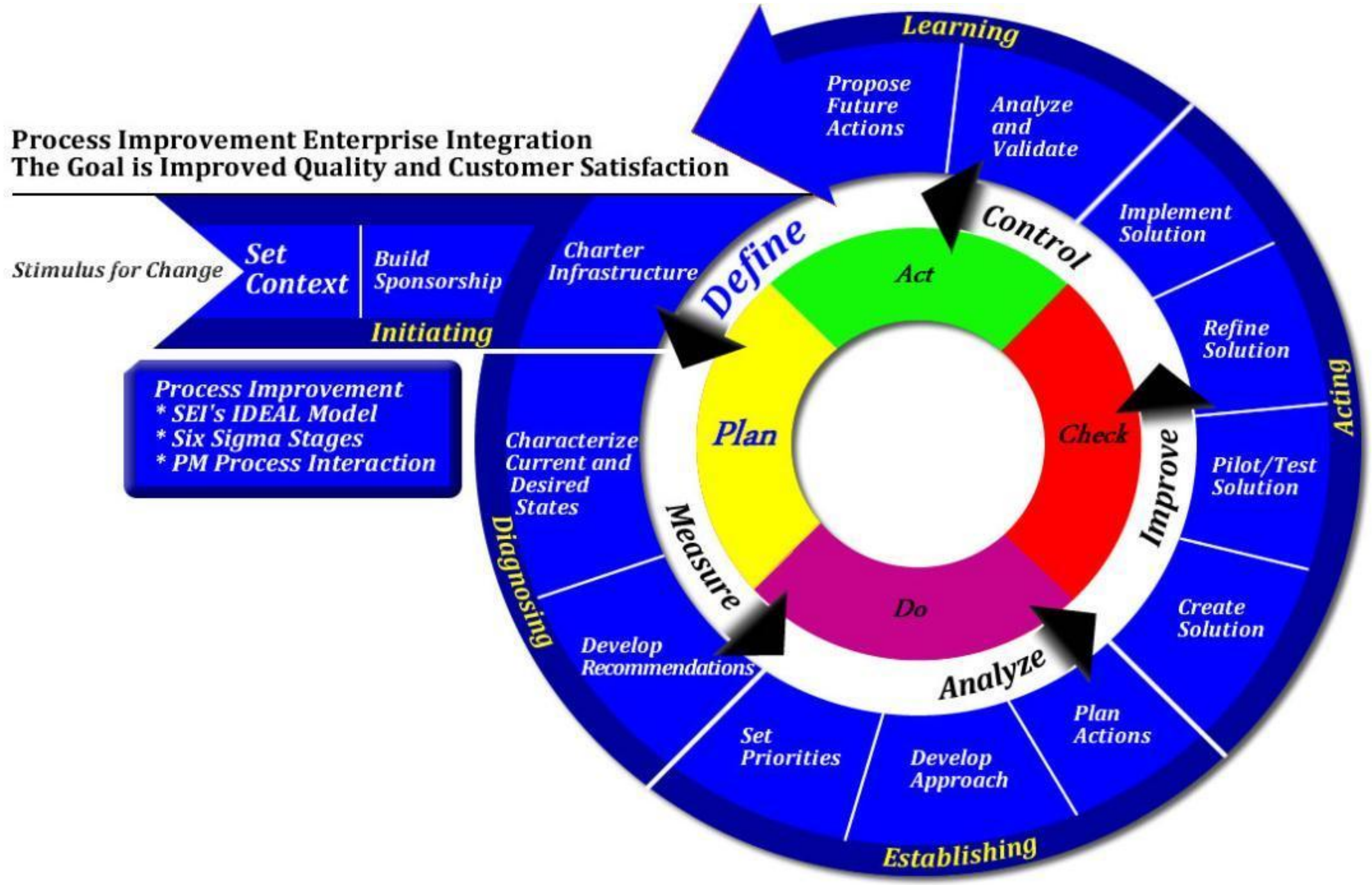
- In the real world, QMS are a mix of best practice & common sense
- Most of the times, they are implemented partially or “almost” followed

Capability Maturity Model Integration (CMMI)


Characteristics of the Maturity levels
Level 5
Optimizing
Focus on process improvement
Level 4
Quantitatively Managed
Processes measured and controlled
Level 3
Defined
Processes characterized for the organization and is proactive.
(Projects tailor their processes from organization's standards)
Level 2
Managed
Processes characterized for projects and is often reactive.
Level 1
Initial
Processes unpredictable, poorly controlled and reactive

Source: https://en.wikipedia.org/wiki/Capability_Maturity_Model_Integration

Software Process Improvement (SPI)



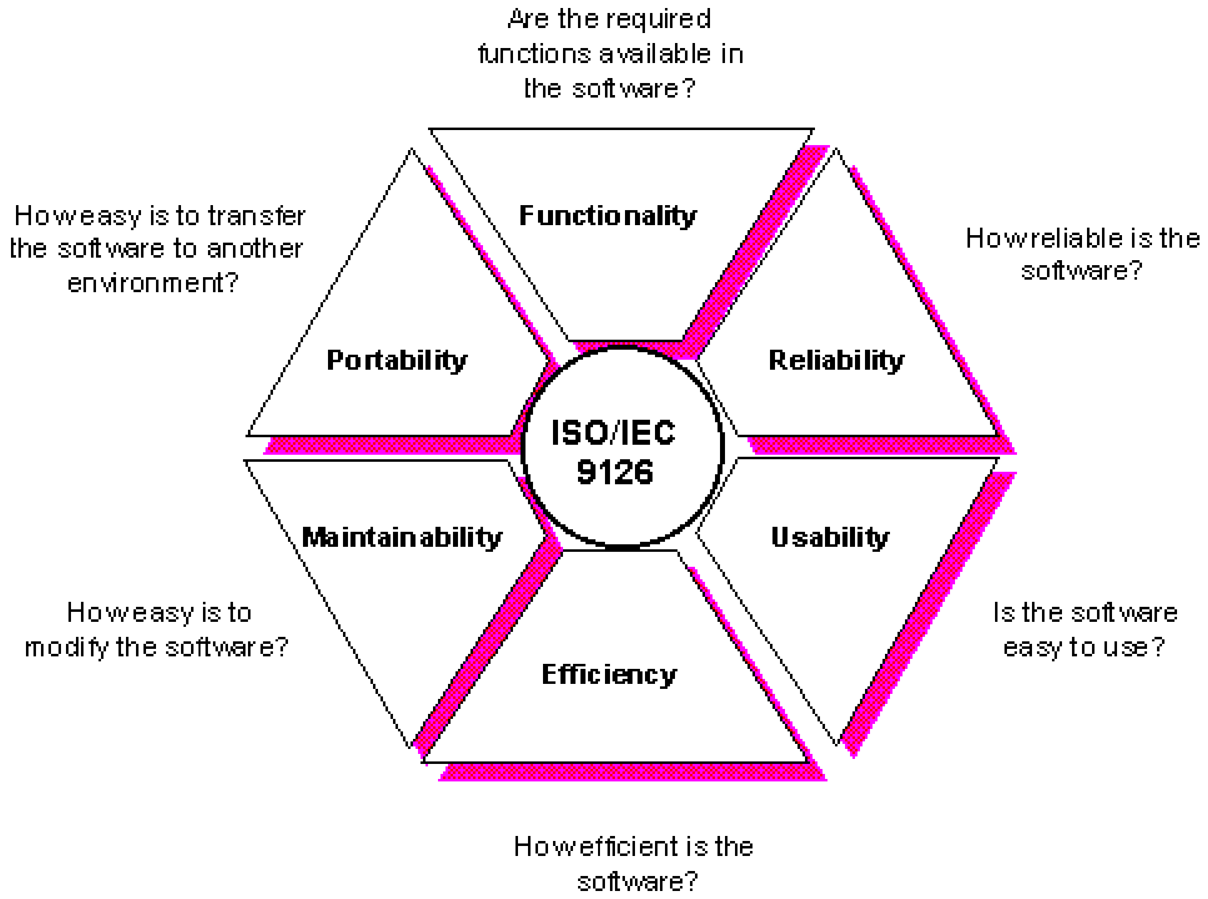
Source: http://www.itcssolutions.com/ITConsulting/procMng.aspx

Software Quality (ISO-CMMI)

- **Quality Criteria**:
  - correctness
  - efficiency
  - flexibility
  - integrity
  - interoperability
  - maintainability
  - portability
  - reliability
  - reusability
  - testability
  - usability
  - ...

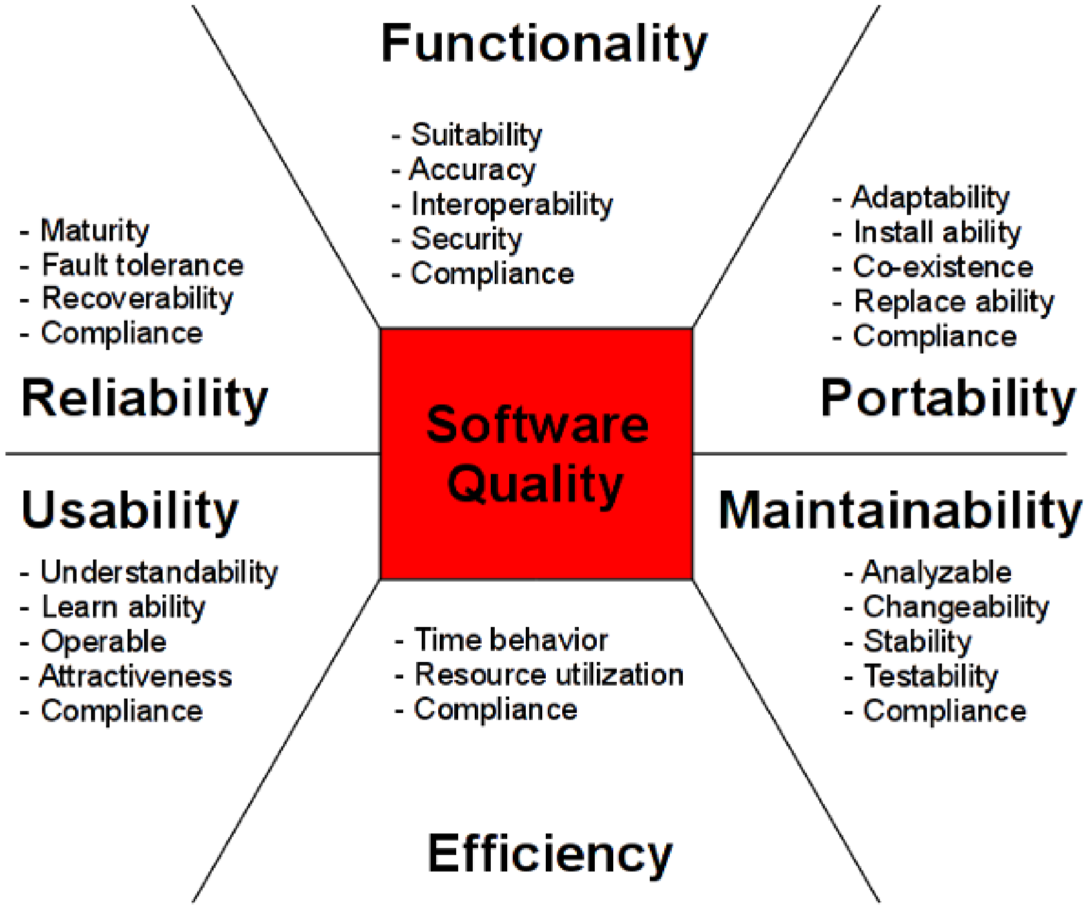
- Some examples:
  - ISO 9001
  - CMM
    - CMU SEI, 5 levels
  - SPICE
    - Developing a standard for software process assessment
    - ISO joint committee, Europe, Australia
  - IEEE 1074, IEEE 12207

Six main quality characteristics (ISO 9126)


Are the required functions available in the software?
Functionality
How easy is to transfer the software to another environment?
Portability
How reliable is the software?
Reliability
ISO/IEC 9126
Maintainability
Usability
How easy is to modify the software?
Is the software easy to use?
Efficiency
How efficient is the software?

Source: https://www.win.tue.nl/~wstomv/edu/2ip30/references/9126ref.html

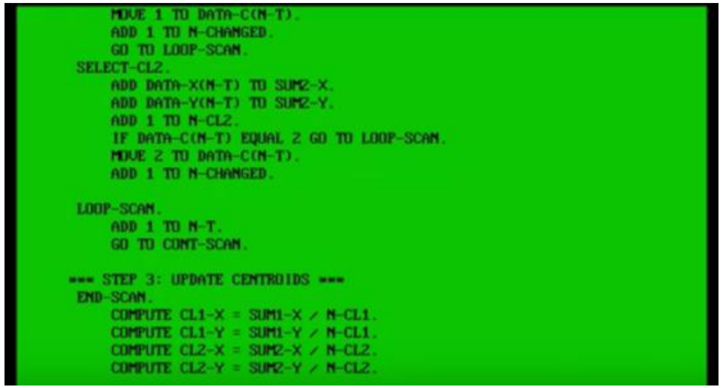
Six main quality characteristics (ISO 9126)



Source: https://rickrainerludwig.wordpress.com/2013/12/16/software-quality-charateristics-in-iso-9126/

Software Metrics

- Code
  - Static
  - Dynamic
- Programmer productivity
- Design
- Testing
- Maintainability
- Management
  - Cost
  - Duration, time
  - Staffing



```
        MOVE 1 TO DATA-C(N-T).
        ADD 1 TO N-CHANGED.
        GO TO LOOP-SCAN.
    SELECT-CL2.
        ADD DATA-X(N-T) TO SUM2-X.
        ADD DATA-Y(N-T) TO SUM2-Y.
        ADD 1 TO N-CL2.
        IF DATA-C(N-T) EQUAL 2 GO TO LOOP-SCAN.
        MOVE 2 TO DATA-C(N-T).
        ADD 1 TO N-CHANGED.

    LOOP-SCAN.
        ADD 1 TO N-T.
        GO TO CONT-SCAN.

   *** STEP 3: UPDATE CENTROIDS ***
    END-SCAN.
        COMPUTE CL1-X = SUM1-X / N-CL1.
        COMPUTE CL1-Y = SUM1-Y / N-CL1.
        COMPUTE CL2-X = SUM2-X / N-CL2.
        COMPUTE CL2-Y = SUM2-Y / N-CL2.
```

Code Metrics

- Estimate number of bugs left in code.
  - From static analysis of code
  - From dynamic execution
- Estimate future failure times: operational reliability

Software Metrics

- Product vs. process
- Most metrics are indirect:
  - No way to measure property directly or
  - Final product does not yet exist
- For predicting, need a model of relationship of predicted variable with other measurable variables.
- Three assumptions (Kitchenham)
  1. We can accurately measure some property of software or process.
  2. A relationship exists between what we can measure and what we want to know.
  3. This relationship is understood, has been validated, and can be expressed in terms of a formula or model.
- Few metrics have been demonstrated to be predictable or related to product or process attributes.

Software Requirements Metrics

- Fairly primitive and predictive power limited.
- Function Points
  - Count number of inputs and output, user interactions, external interfaces, files used.
  - Assess each for complexity and multiply by a weighting factor.
  - Used to predict size or cost and to assess project productivity.
- Number of requirements errors found (to assess quality)
- Change request frequency
  - To assess stability of requirements.
  - Frequency should decrease over time. If not, requirements analysis may not have been done properly.

Software Design Metrics

- Number of parameters
  - Tries to capture coupling between modules.
  - Understanding modules with large number of parameters will require more time and effort (assumption).
  - Modifying modules with large number of parameters likely to have side effects on other modules.

- Number of modules.

- Number of modules called (estimating complexity of maintenance).

  Fan-in: number of modules that call a particular module.
  Fan-out: how many other modules it calls.

  - High fan-in means many modules depend on this module.
  - High fan-out means module depends on many other modules.

    Makes understanding harder and maintenance more time-consuming.

Software Design Metrics

- Cohesion metric
  - Construct flow graph for module.
    - Each vertex is an executable statement.
    - For each node, record variables referenced in statement.
  - Determine how many independent paths of the module go through the different statements.
    - If a module has high cohesion, most of variables will be used by statements in most paths.
    - Highest cohesion is when all the independent paths use all the variables in the module.

Programmer Effort Metrics

- Because software intangible, not possible to measure directly.
- If poor quality software produced quickly, may appear to be more productive than if produce reliable and easy to maintain software (measure only over software development phase).
  - More does not always mean better.
  - May ultimately involve increased system maintenance costs.
- Common measures:
  - Lines of source code written per programmer month.
  - Object instructions produced per programmer month.
  - Pages of documentation written per programmer month.
  - Test cases written and executed per programmer month.

Programmer Effort Metrics (*cont.*)

- Take total number of source code lines delivered and divide by total time required to complete project.
  - What is a source line of code? (declarations? comments? macros?)
  - How treat source lines containing more than a single statement?
  - More productive when use assembly language? (the more expressive the language, the lower the apparent productivity)
  - All tasks subsumed under coding task although coding time represents small part of time needed to complete a project.

- Use number of object instructions generated.
  - More objective.
  - Difficult to estimate until code actually produced.
  - Amount of object code generated dependent on high-level language programming style.

Programmer Effort Metrics (*cont.*)

- Using pages of documentation penalizes writers who take time to express themselves clearly and concisely.

So difficult to give average figure. *LOC metrics*

- For large, embedded system may be as low as 30 lines/programmer-month.
- Simple business systems may be 600 lines.
- Studies show great variability in individual productivity. Best are twenty times more productive than worst.

→ *"What about visual programming?"* (e.g. Web apps)
→ *"What about non-programming effort?"* (e.g. data analytics)
→ *"What about maintaining a codebase?"* (few changes in LOC)

Static Code Analysis – Problems...

- Doesn't change as program changes.
- High correlation with program size.
- No real intuitive reason for many of metrics.
- Ignores many factors: e.g., computing environment, application area, particular algorithms implemented, characteristics of users, ability of programmers,.
- Very easy to get around. Programmers may introduce more obscure complexity in order to minimize properties measured by particular complexity metric.

- Size is best predictor of inherent faults remaining at start of program test.
- One study has shown that besides size, 3 significant additional factors:

1. Specification change activity, measured in pages of specification changes per k lines of code.
2. Average programmer skill, measured in years.
3. Thoroughness of design documentation, measured in pages of developed (new plus modified) design documents per k lines of code.

Bug estimations using Dynamic Analysis

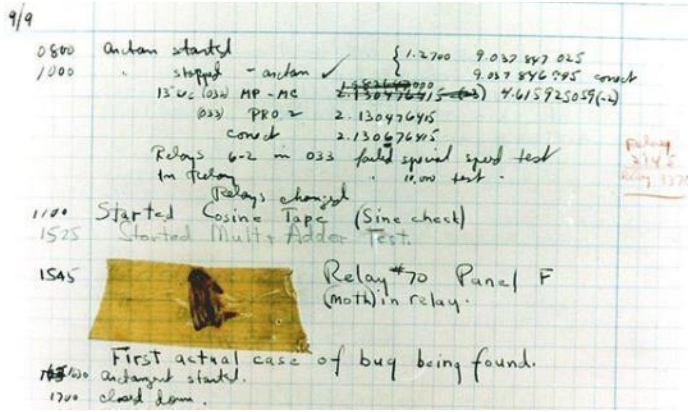
- Estimate number remaining from number found.
  - Failure count models
  - Error seeding models
- Assumptions:
  - Seeded faults equivalent to inherent faults in difficulty of detection.
  - A direct relationship between characteristics and number of exposed and undiscovered faults.
  - Unreliability of system will be directly proportional to number of faults that remain.
  - A constant rate of fault detection.

- What does an estimate of remaining errors mean?

  Interested in performance of program, not in how many bugs it contains.

- Most requirements written in terms of operational reliability, not number of bugs.

- Alternative is to estimate failure rates or future interfailure times.

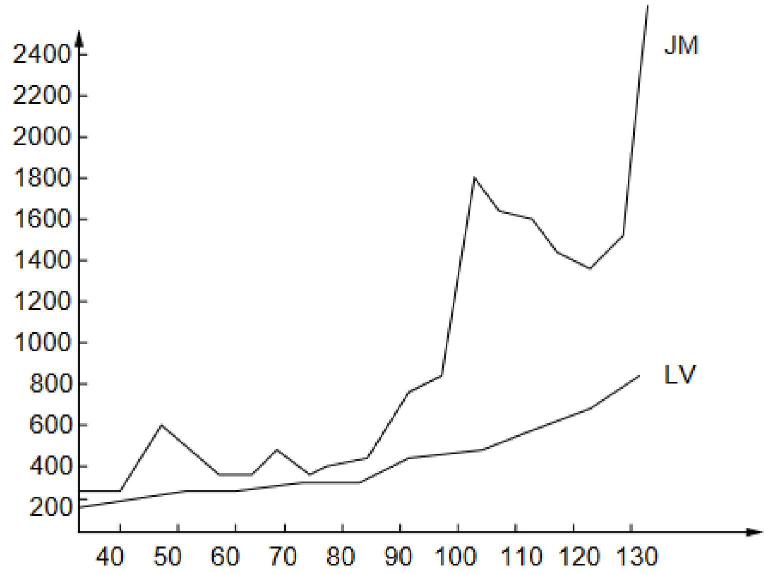


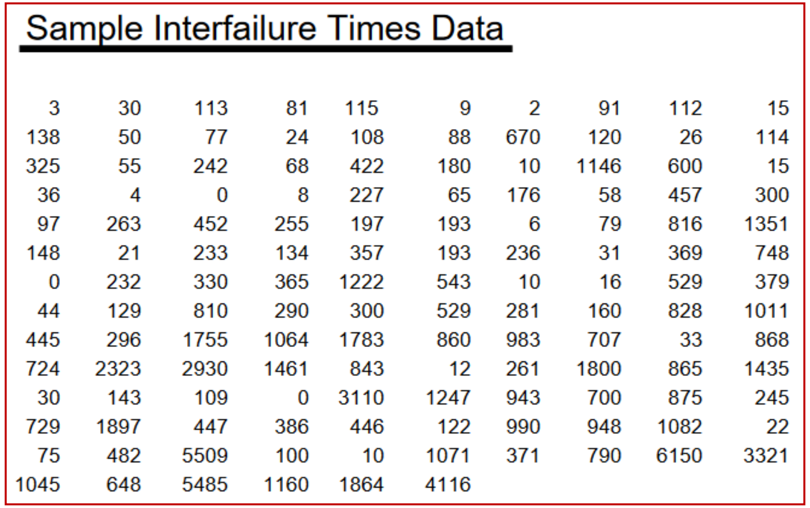
9/9

0800 Antan started
1000 " stopped - antan ✓ {1.2700 9.037 847 025
9.037 846 995 conct
13" UC (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (-3) 4.615925059(-2)
(033) PRO 2 2.130476415
conct 2.130676415
Relays 6-2 in 033 failed special speed test
In telay 11,000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F (moth) in relay.
First actual case of bug being found.
1630 antangent started.
1700 closed down.

Relay 2145
Relay 3370

Dynamic Analysis: Stochastic models





Sample Interfailure Times Data

3	30	113	81	115	9	2	91	112	15
138	50	77	24	108	88	670	120	26	114
325	55	242	68	422	180	10	1146	600	15
36	4	0	8	227	65	176	58	457	300
97	263	452	255	197	193	6	79	816	1351
148	21	233	134	357	193	236	31	369	748
0	232	330	365	1222	543	10	16	529	379
44	129	810	290	300	529	281	160	828	1011
445	296	1755	1064	1783	860	983	707	33	868
724	2323	2930	1461	843	12	261	1800	865	1435
30	143	109	0	3110	1247	943	700	875	245
729	1897	447	386	446	122	990	948	1082	22
75	482	5509	100	10	1071	371	790	6150	3321
1045	648	5485	1160	1864	4116				

- Different models can give varying results for the same data; there is no way to know a priori which model will provide the best results in a given situation.

- "The nature of the software engineering process is too poorly understood to provide a basis for selecting a particular model."

Software Management Metrics

- Techniques for software cost estimation

    1. Algorithmic cost modeling:
        - Model developed using historical cost information that relates some software metric (usually lines of code) to project cost.
        - Estimate made of metric and then model predicts effort required.
        - The most scientific approach but not necessarily the most accurate.
    2. Expert judgement
    3. Estimation by analogy: useful when other projects in same domain have been completed.

Software Management Metrics (*cont.*)

4. Parkinson's Law: Work expands to fill the time available.
   - Cost is determined by available resources
   - If software has to be developed in 12 months and you have 5 people available, then effort required is estimated to be 60 person months.

5. Pricing to win: estimated effort based on customer's budget.

6. Top-down estimation: cost estimate made by considering overall function and how functionality provided by interacting sub-functions. Made on basis of logical function rather than the components implementing that function.

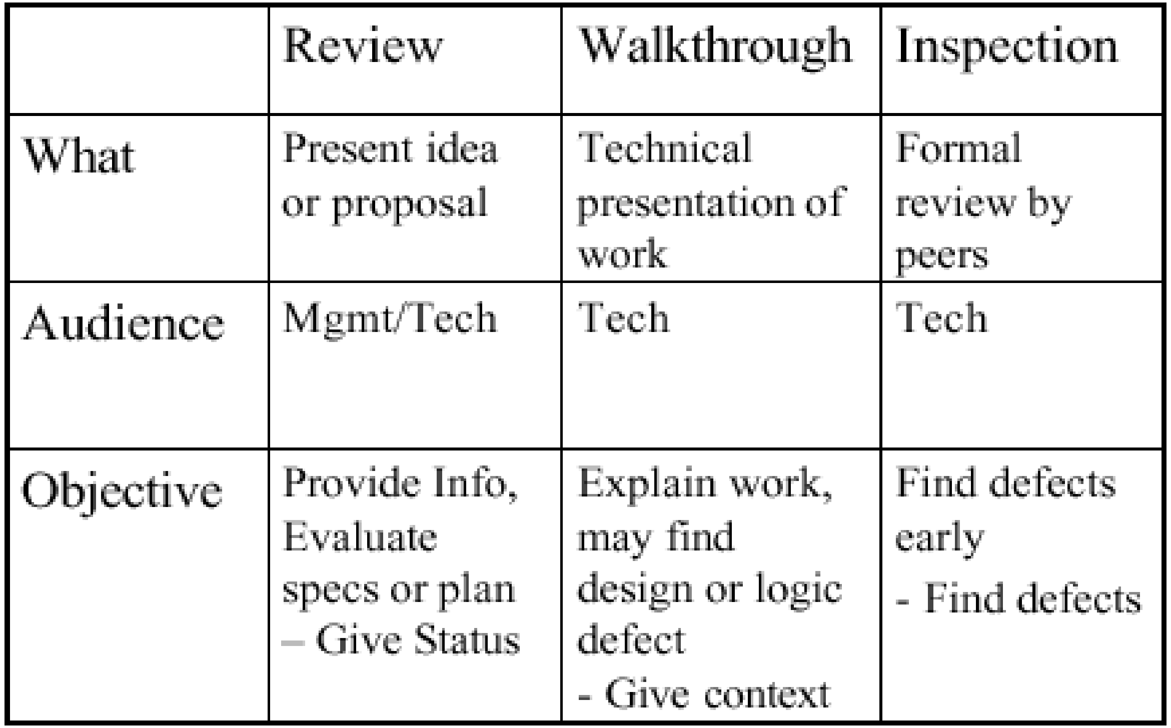
7. Bottom-up function: cost of each component estimated and then added to produce final cost estimate.

Software Reliability

Software Reliability: The probability that a program will perform its specified function for a stated time under specified conditions.

- Execute program until "failure" occurs, the underlying error found and removed (in zero time), and resume execution.
- Use a probability distribution function for the interfailure time (assumed to be a random variable) to predict future times to failure.
- Examining the nature of the sequence of elapsed times from one failure to the next.
- Assumes occurrence of software failures is a stochastic process.

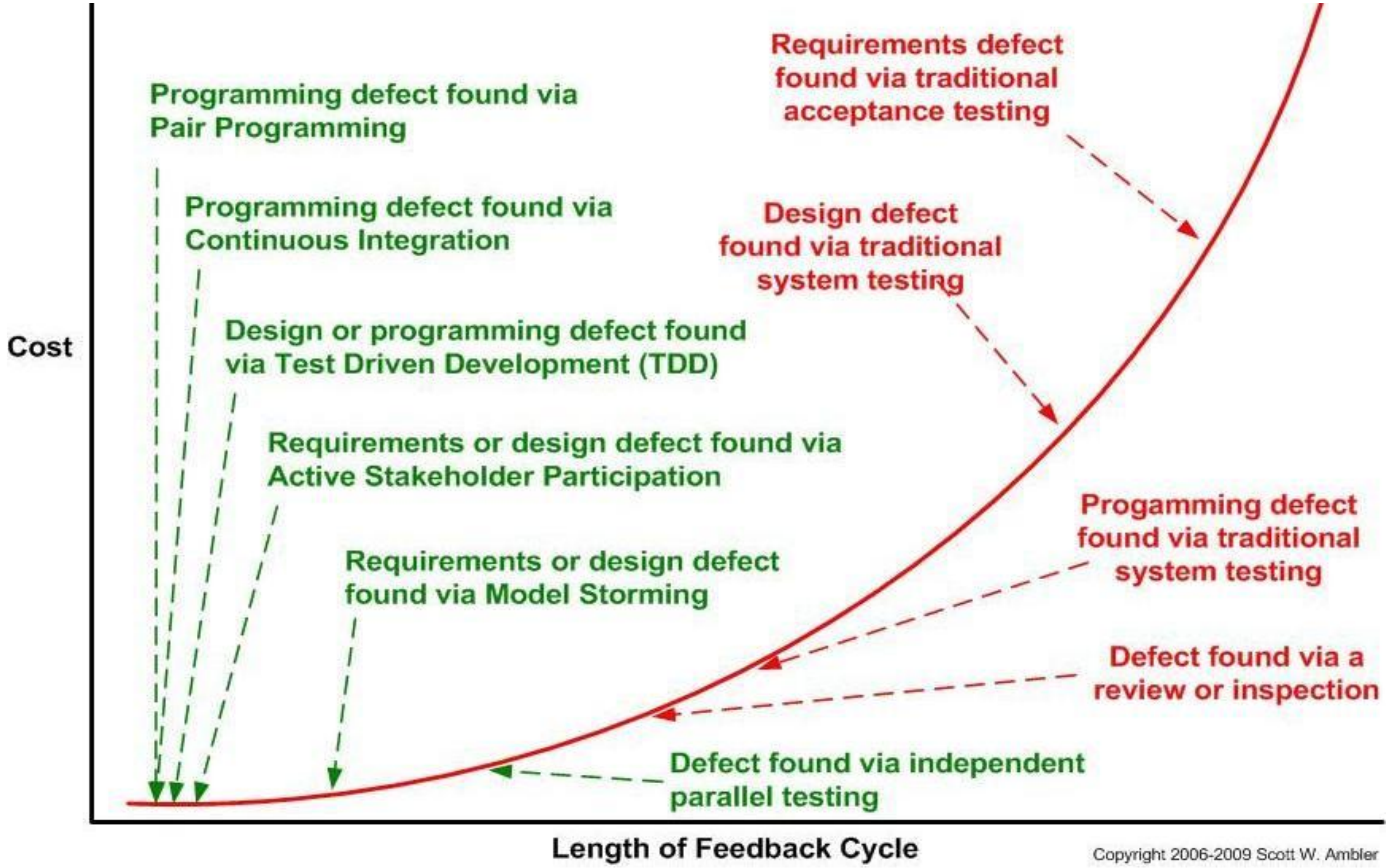
Software reviews, walkthroughs, inspections



	Review	Walkthrough	Inspection
What	Present idea or proposal	Technical presentation of work	Formal review by peers
Audience	Mgmt/Tech	Tech	Tech
Objective	Provide Info, Evaluate specs or plan – Give Status	Explain work, may find design or logic defect - Give context	Find defects early - Find defects

Source: http://www.math.uaa.alaska.edu/~afkjm/cs470/handouts/inspections.pdf

The cost of finding (or missing) bugs



Source: https://dev.astrotech.io/sonarqube/quality-models.html

Worst Software Failures: Mariner I

Navigation Gone Wrong

Mariner 1 Destroyed

The first American spacecraft sent to explore another planet, Mariner 1 was launched on July 22, 1962. But it never reached Venus. It never even reached space.

WHAT HAPPENED

Unbeknownst to its operators, the launch computer that controlled the Atlas rocket carrying Mariner 1 contained a tiny programming error. A single character had been left out of the guidance equations.

THE CONSEQUENCES

About four minutes into its flight, the Atlas rocket carrying Mariner 1 began behaving erratically. The rocket had to be destroyed, and with it Mariner 1.

LESSONS LEARNED

The disaster revealed a critical need to thoroughly debug software before launch. NASA also learned that software can be engineered so that small errors do not impact safety. Thanks to NASA's corrective actions, several Apollo lunar modules safely landed on the Moon despite minor software "bugs."

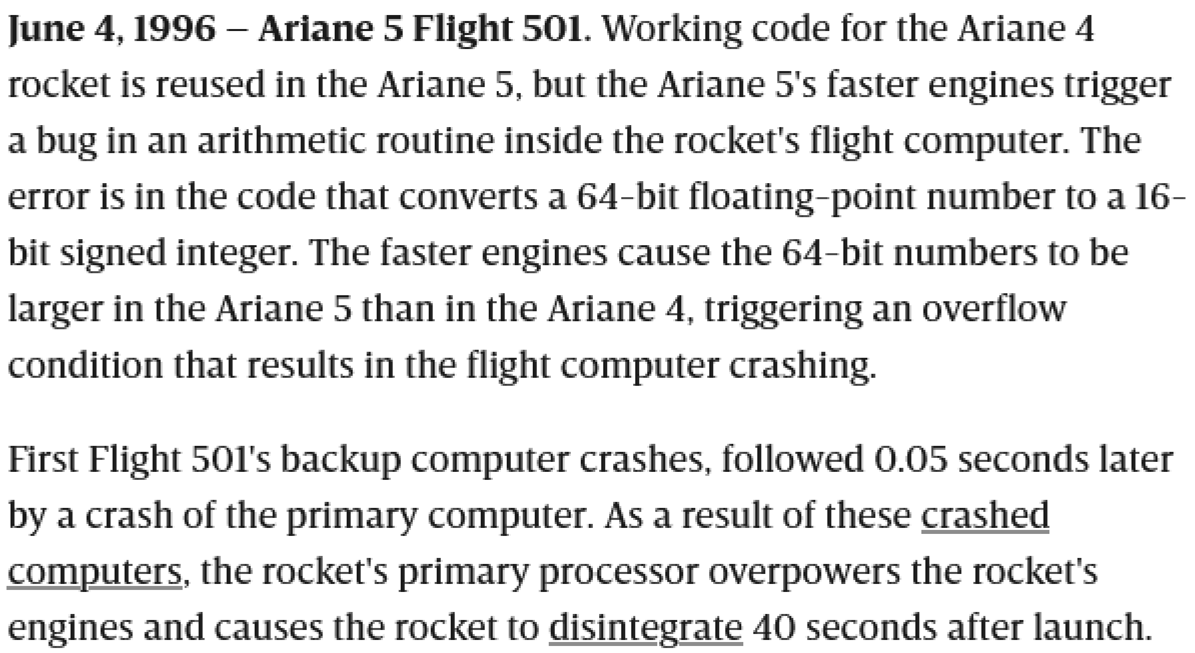
Mariner 1 Launch

Launch of Mariner 1 on July 22, 1962.

Credit: NASA Kennedy Space Center via National Air and Space Museum, Smithsonian Institution

Source: https://timeandnavigation.si.edu/navigating-space/challenges/mariner-1-destroyed

Worst Software Failures: Ariane 5



June 4, 1996 – Ariane 5 Flight 501. Working code for the Ariane 4 rocket is reused in the Ariane 5, but the Ariane 5's faster engines trigger a bug in an arithmetic routine inside the rocket's flight computer. The error is in the code that converts a 64-bit floating-point number to a 16-bit signed integer. The faster engines cause the 64-bit numbers to be larger in the Ariane 5 than in the Ariane 4, triggering an overflow condition that results in the flight computer crashing.

First Flight 501's backup computer crashes, followed 0.05 seconds later by a crash of the primary computer. As a result of these crashed computers, the rocket's primary processor overpowers the rocket's engines and causes the rocket to disintegrate 40 seconds after launch.

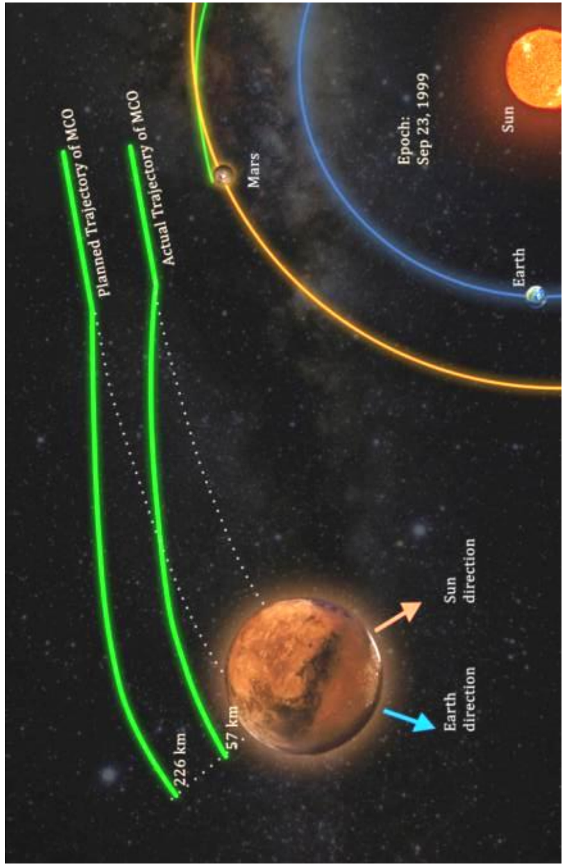
Source: https://www.wired.com/2005/11/historys-worst-software-bugs/

Worst Software Failures: Mars Climate Orbiter

Mars Climate Orbiter built by NASA's Jet Propulsion Laboratory approached the Red Planet at the wrong angle. At this point, it could easily have been renamed the Mars Climate Bright Light in the Upper Atmosphere, and shortly afterward been renamed the Mars Climate Debris Drifting Through the Sky.

There were several problems with this spacecraft -- its uneven payload made it torque during flight, and its project managers neglected some important details during several stages of the mission. But the biggest problem was that different parts of the engineering team were using different units of measurement. One group working on the thrusters measured in English units of pounds-force seconds; the others used metric Newton-seconds. And whoever checked the numbers didn't use the red pen like a pedantic high-school teacher.

The result: The thrusters were 4.45 times more powerful than they should have been. If this goof had been spotted earlier, it could have been compensated for, but it wasn't, and the result of that inattention is now lost in space, possibly in pieces.



Source: https://www.computerworld.com/article/2515483/epic-failures-11-infamous-software-bugs.html

Worst Software Failures: ESA Schiaparelli

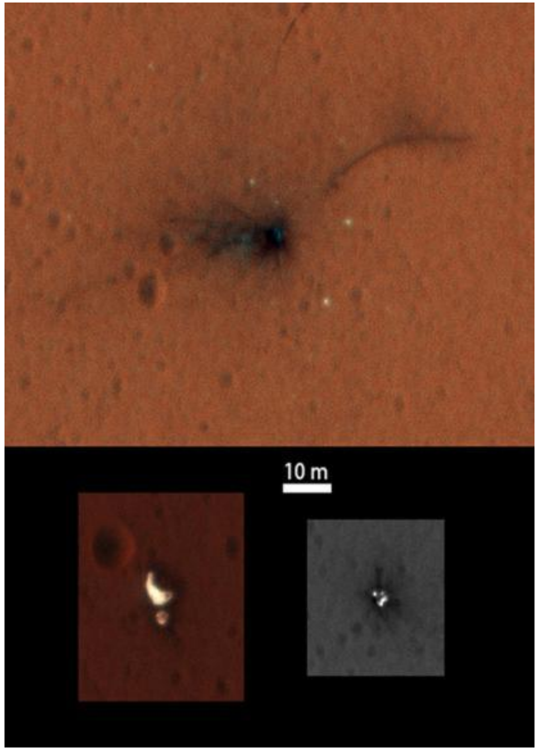
European Mars Lander Crashed Due to Data Glitch, ESA Concludes

By Mike Wall published May 27, 2017

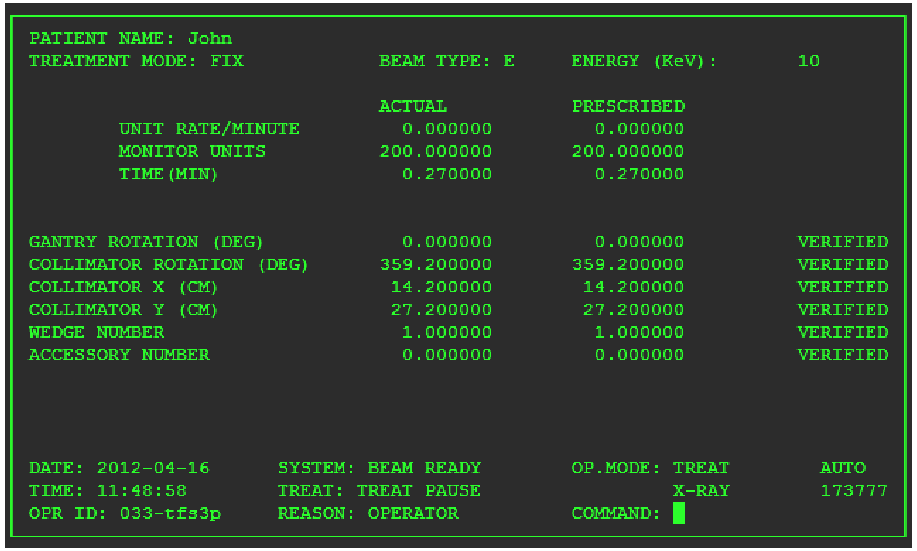
The reason Europe's Schiaparelli Mars lander failed to touch down safely last fall is that conflicting data confused the craft's onboard computer, according to the newly completed crash investigation.

Things started to go wrong for Schiaparelli about 3 minutes after it hit the Martian atmosphere on Oct. 19, 2016. At that time, the lander deployed its parachute and then began spinning unexpectedly fast, according to the investigation, which concluded last week.

This superfast rotation briefly saturated Schiaparelli's spin-measuring instrument, which "resulted in a large attitude-estimation error by the guidance, navigation and control-system software," European Space Agency (ESA) officials wrote in a statement Wednesday (May 24). ("Attitude" refers to a spacecraft's orientation.) [In Photos: Europe's Schiaparelli Mars Landing Day]



Worst Software Failures: Therac-25



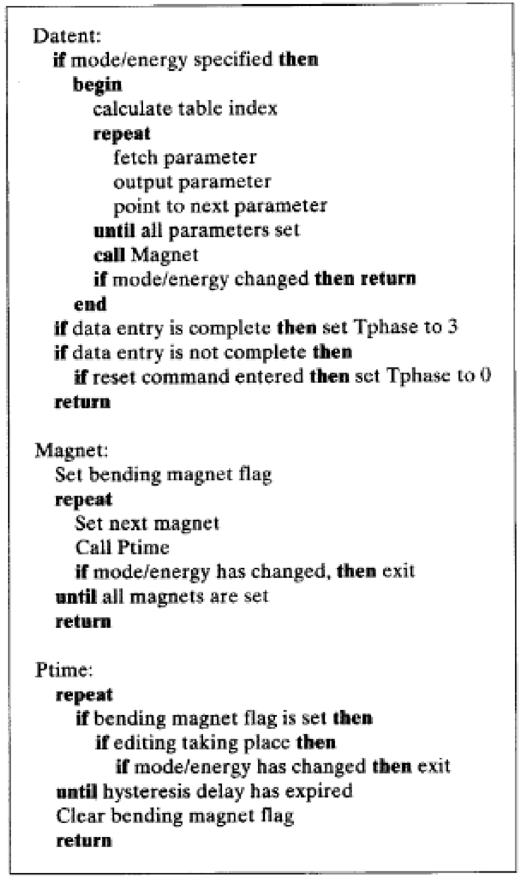
```
PATIENT NAME: John
TREATMENT MODE: FIX              BEAM TYPE: E     ENERGY (KeV):       10

                                 ACTUAL           PRESCRIBED
        UNIT RATE/MINUTE           0.000000         0.000000
        MONITOR UNITS            200.000000       200.000000
        TIME(MIN)                  0.270000         0.270000


GANTRY ROTATION (DEG)              0.000000         0.000000         VERIFIED
COLLIMATOR ROTATION (DEG)        359.200000       359.200000         VERIFIED
COLLIMATOR X (CM)                 14.200000        14.200000         VERIFIED
COLLIMATOR Y (CM)                 27.200000        27.200000         VERIFIED
WEDGE NUMBER                       1.000000         1.000000         VERIFIED
ACCESSORY NUMBER                   0.000000         0.000000         VERIFIED




DATE: 2012-04-16      SYSTEM: BEAM READY        OP.MODE: TREAT        AUTO
TIME: 11:48:58        TREAT: TREAT PAUSE                 X-RAY        173777
OPR ID: 033-tfs3p     REASON: OPERATOR          COMMAND: █
```



```
Datent:
  if mode/energy specified then
    begin
      calculate table index
      repeat
        fetch parameter
        output parameter
        point to next parameter
      until all parameters set
      call Magnet
      if mode/energy changed then return
    end
  if data entry is complete then set Tphase to 3
  if data entry is not complete then
    if reset command entered then set Tphase to 0
  return

Magnet:
  Set bending magnet flag
  repeat
    Set next magnet
    Call Ptime
    if mode/energy has changed, then exit
  until all magnets are set
  return

Ptime:
  repeat
    if bending magnet flag is set then
      if editing taking place then
        if mode/energy has changed then exit
  until hysteresis delay has expired
  Clear bending magnet flag
  return
```

Figure 3. Datent, Magnet, and Ptime subroutines.

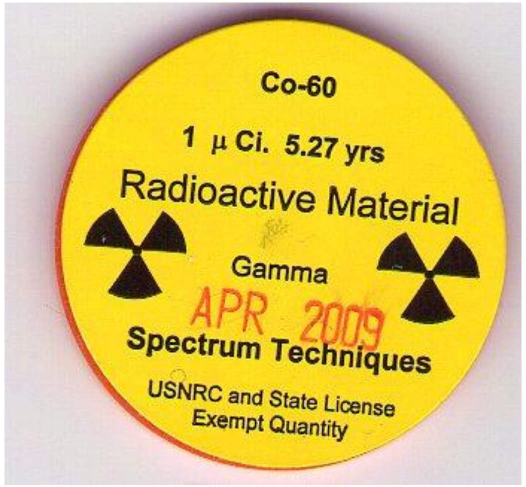
- Radiation therapy machine operated in USA and Canada.
- At least six overdose incidents in patients (1985-1987).
- Cause: Concurrency error due to race conditions.
- "Refurbished" PDP-11 assembly, single-person project.

See also: "An investigation of the Therac-25 accidents", IEEE Computer, 1993.

See also: https://www.youtube.com/watch?v=Ap0orGCiou8

Source: https://en.wikipedia.org/wiki/Therac-25

Worst Software Failures: Multidata/Cobalt-60



November 2000 – National Cancer Institute, Panama City. In a series of accidents, therapy planning software created by Multidata Systems International, a U.S. firm, miscalculates the proper dosage of radiation for patients undergoing radiation therapy.

Multidata's software allows a radiation therapist to draw on a computer screen the placement of metal shields called "blocks" designed to protect healthy tissue from the radiation. But the software will only allow technicians to use four shielding blocks, and the Panamanian doctors wish to use five.

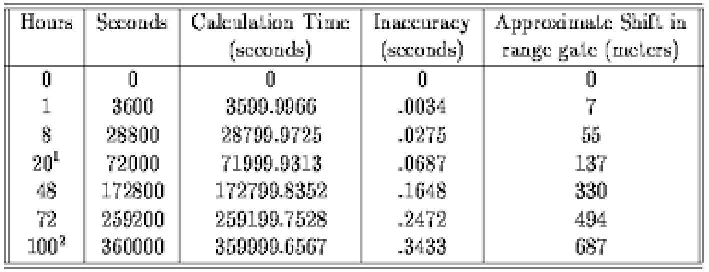
The doctors discover that they can trick the software by drawing all five blocks as a single large block with a hole in the middle. What the doctors don't realize is that the Multidata software gives different answers in this configuration depending on how the hole is drawn: draw it in one direction and the correct dose is calculated, draw in another direction and the software recommends twice the necessary exposure.

Source: https://www.wired.com/2005/11/historys-worst-software-bugs/

Worst Software Failures: Patriot missiles

Patriot missile mistiming

During the first Persian Gulf war, Iraqi-fired Scud missiles were the most threatening airborne enemies to U.S. troops. Once one of these speeding death rockets launched, the U.S.'s best defense was to intercept it with an antiballistic Patriot missile. The Patriot worked a bit like a shotgun, getting within range of an oncoming missile before blasting out a cloud of 1,000 pellets to detonate its warhead.

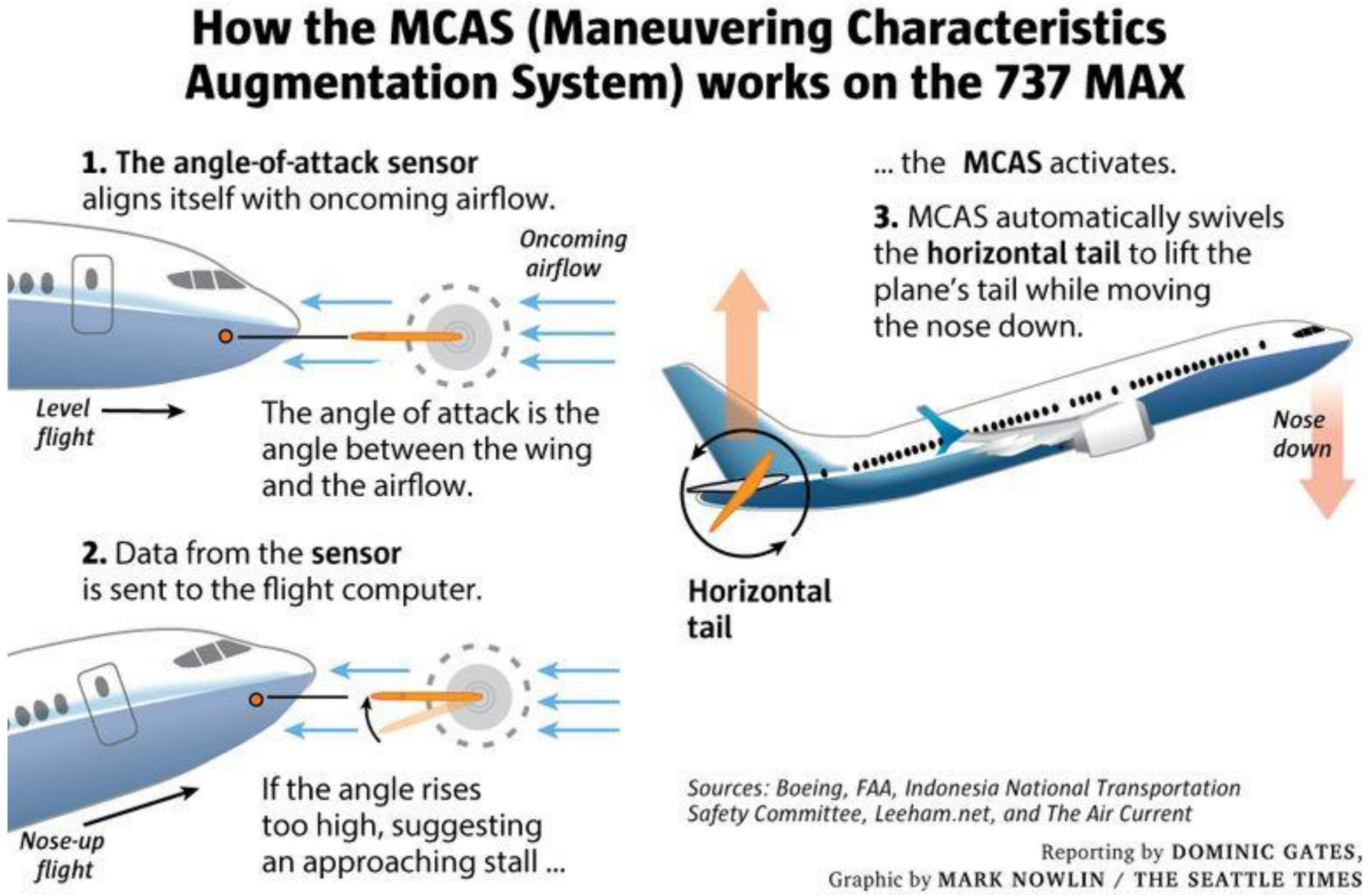


Hours	Seconds	Calculation Time (seconds)	Inaccuracy (seconds)	Approximate Shift in range gate (meters)
0	0	0	0	0
1	3600	3599.9966	.0034	7
8	28800	28799.9725	.0275	55
20[1]	72000	71999.9313	.0687	137
48	172800	172799.8352	.1648	330
72	259200	259199.7528	.2472	494
100[2]	360000	359999.6567	.3433	687

A Patriot needed to deploy its pellets between 5 and 10 meters from an oncoming missile for the best results. This requires split-second timing, which is always tricky with two objects moving very fast toward each other. Even the Patriot's most prominent booster, then-President George H.W. Bush, conceded that
one Scud (out of 42 fired) got past the Patriot. The single failure the president acknowledged was at a U.S. base in Dhahran, Saudi Arabia, on Feb. 25, 1991, and it cost 28 soldiers their lives. The fault was traced to a software error.

Source: https://www.computerworld.com/article/2515483/epic-failures-11-infamous-software-bugs.html

Worst Software Failures: Boeing 737 MAX



How the MCAS (Maneuvering Characteristics Augmentation System) works on the 737 MAX

1. The angle-of-attack sensor aligns itself with oncoming airflow.

The angle of attack is the angle between the wing and the airflow.

2. Data from the **sensor** is sent to the flight computer.

If the angle rises too high, suggesting an approaching stall ...

... the **MCAS** activates.

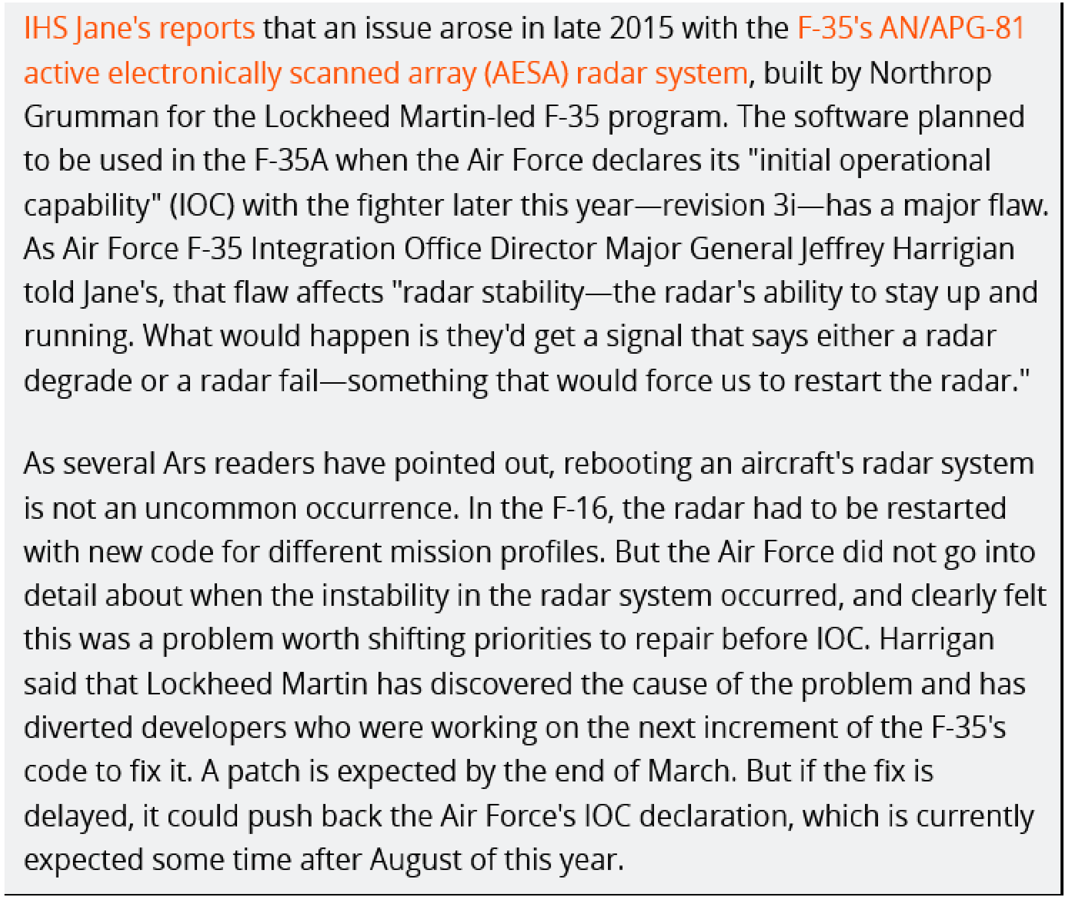
3. MCAS automatically swivels the **horizontal tail** to lift the plane's tail while moving the nose down.

Sources: Boeing, FAA, Indonesia National Transportation Safety Committee, Leeham.net, and The Air Current

Reporting by DOMINIC GATES, Graphic by MARK NOWLIN / THE SEATTLE TIMES

Source: https://www.seattletimes.com/seattle-news/times-watchdog/the-inside-story-of-mcas-how-boeings-737-max-system-gained-power-and-lost-safeguards/

Worst Software Failures: F-35 AESA radar

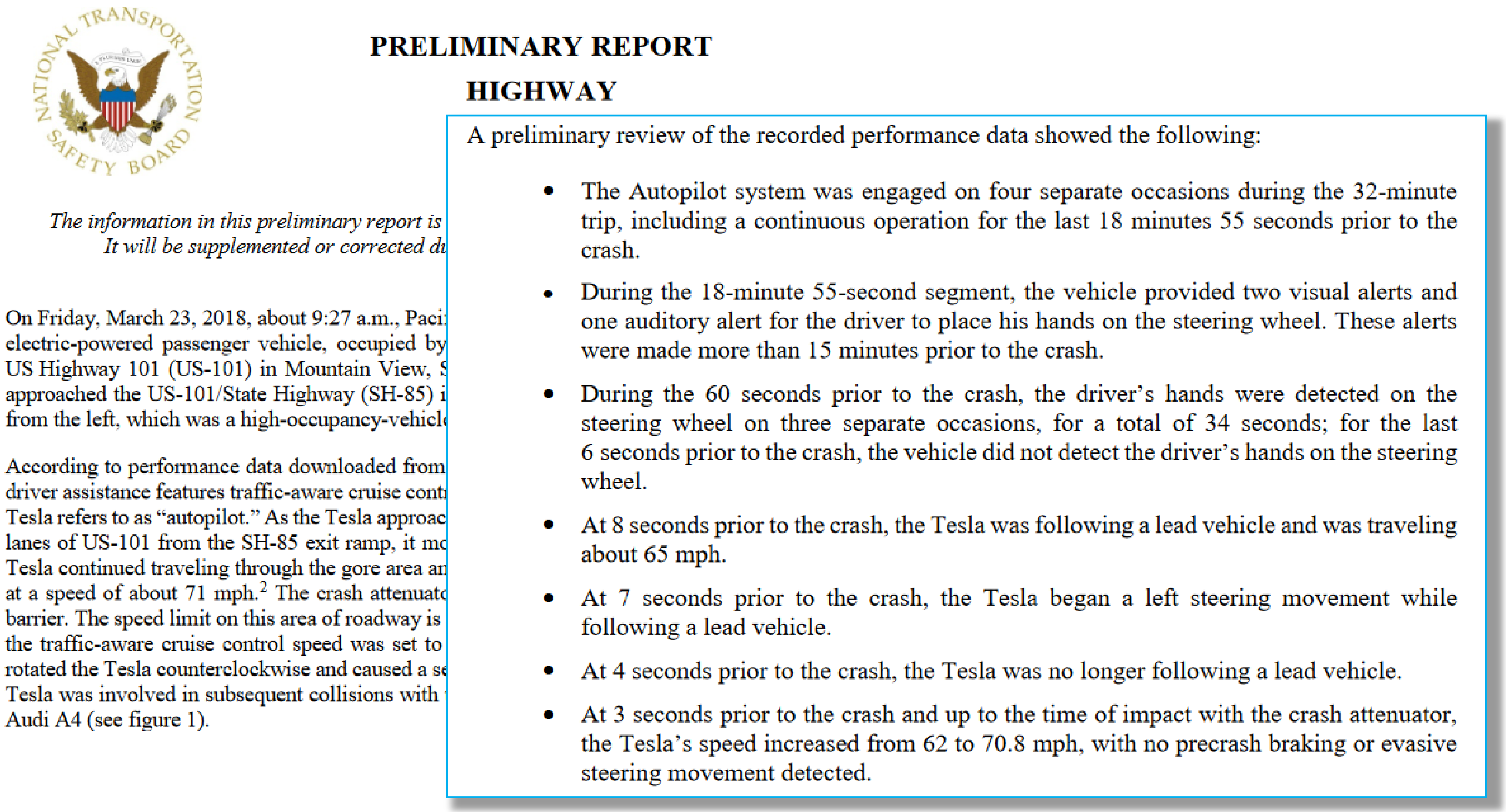


IHS Jane's reports that an issue arose in late 2015 with the F-35's AN/APG-81 active electronically scanned array (AESA) radar system, built by Northrop Grumman for the Lockheed Martin-led F-35 program. The software planned to be used in the F-35A when the Air Force declares its "initial operational capability" (IOC) with the fighter later this year—revision 3i—has a major flaw. As Air Force F-35 Integration Office Director Major General Jeffrey Harrigian told Jane's, that flaw affects "radar stability—the radar's ability to stay up and running. What would happen is they'd get a signal that says either a radar degrade or a radar fail—something that would force us to restart the radar."

As several Ars readers have pointed out, rebooting an aircraft's radar system is not an uncommon occurrence. In the F-16, the radar had to be restarted with new code for different mission profiles. But the Air Force did not go into detail about when the instability in the radar system occurred, and clearly felt this was a problem worth shifting priorities to repair before IOC. Harrigan said that Lockheed Martin has discovered the cause of the problem and has diverted developers who were working on the next increment of the F-35's code to fix it. A patch is expected by the end of March. But if the fix is delayed, it could push back the Air Force's IOC declaration, which is currently expected some time after August of this year.

Source: https://arstechnica.com/information-technology/2016/03/f-35-radar-system-has-bug-that-requires-hard-reboot-in-flight/

Worst Software Failures: Tesla car crash



PRELIMINARY REPORT

HIGHWAY

The information in this preliminary report is
It will be supplemented or corrected du

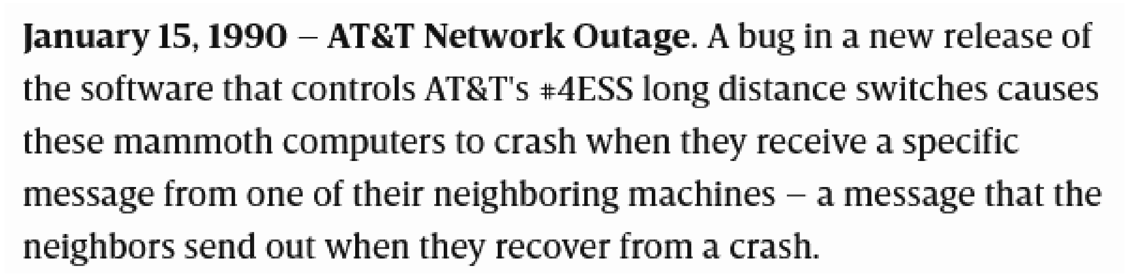
On Friday, March 23, 2018, about 9:27 a.m., Paci
electric-powered passenger vehicle, occupied by
US Highway 101 (US-101) in Mountain View, S
approached the US-101/State Highway (SH-85) i
from the left, which was a high-occupancy-vehicl

According to performance data downloaded from
driver assistance features traffic-aware cruise cont
Tesla refers to as "autopilot." As the Tesla approac
lanes of US-101 from the SH-85 exit ramp, it mo
Tesla continued traveling through the gore area an
at a speed of about 71 mph.[2] The crash attenuato
barrier. The speed limit on this area of roadway is
the traffic-aware cruise control speed was set to
rotated the Tesla counterclockwise and caused a se
Tesla was involved in subsequent collisions with t
Audi A4 (see figure 1).

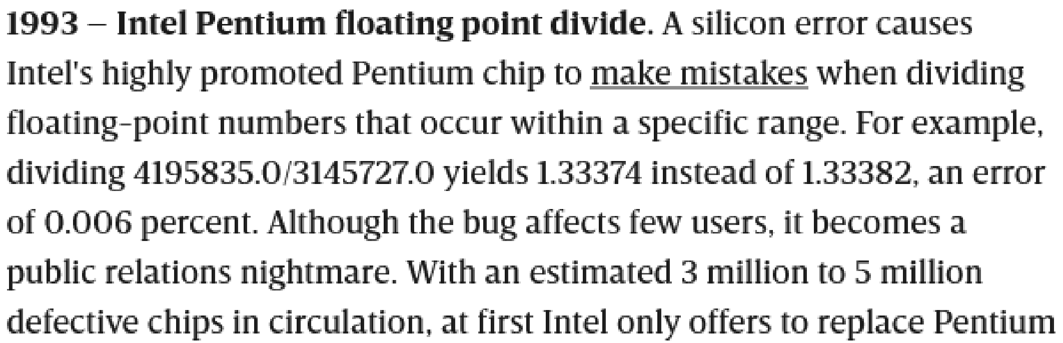
A preliminary review of the recorded performance data showed the following:

- The Autopilot system was engaged on four separate occasions during the 32-minute trip, including a continuous operation for the last 18 minutes 55 seconds prior to the crash.
- During the 18-minute 55-second segment, the vehicle provided two visual alerts and one auditory alert for the driver to place his hands on the steering wheel. These alerts were made more than 15 minutes prior to the crash.
- During the 60 seconds prior to the crash, the driver's hands were detected on the steering wheel on three separate occasions, for a total of 34 seconds; for the last 6 seconds prior to the crash, the vehicle did not detect the driver's hands on the steering wheel.
- At 8 seconds prior to the crash, the Tesla was following a lead vehicle and was traveling about 65 mph.
- At 7 seconds prior to the crash, the Tesla began a left steering movement while following a lead vehicle.
- At 4 seconds prior to the crash, the Tesla was no longer following a lead vehicle.
- At 3 seconds prior to the crash and up to the time of impact with the crash attenuator, the Tesla's speed increased from 62 to 70.8 mph, with no precrash braking or evasive steering movement detected.

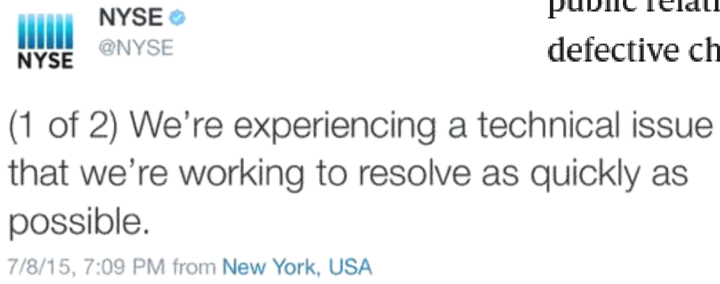
Worst Software Failures: *(and the list goes on…)*



January 15, 1990 – AT&T Network Outage. A bug in a new release of the software that controls AT&T's #4ESS long distance switches causes these mammoth computers to crash when they receive a specific message from one of their neighboring machines – a message that the neighbors send out when they recover from a crash.



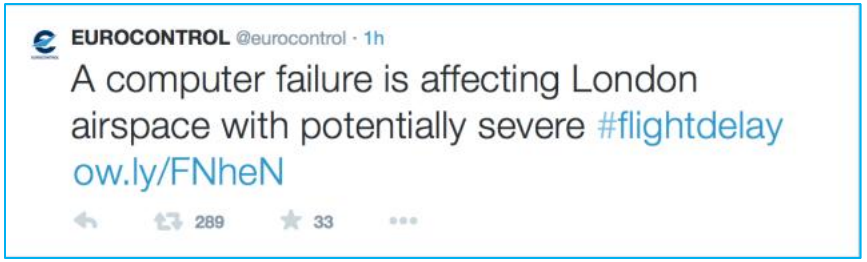
1993 – Intel Pentium floating point divide. A silicon error causes Intel's highly promoted Pentium chip to make mistakes when dividing floating-point numbers that occur within a specific range. For example, dividing 4195835.0/3145727.0 yields 1.33374 instead of 1.33382, an error of 0.006 percent. Although the bug affects few users, it becomes a public relations nightmare. With an estimated 3 million to 5 million defective chips in circulation, at first Intel only offers to replace Pentium



NYSE
@NYSE

(1 of 2) We're experiencing a technical issue that we're working to resolve as quickly as possible.

7/8/15, 7:09 PM from New York, USA



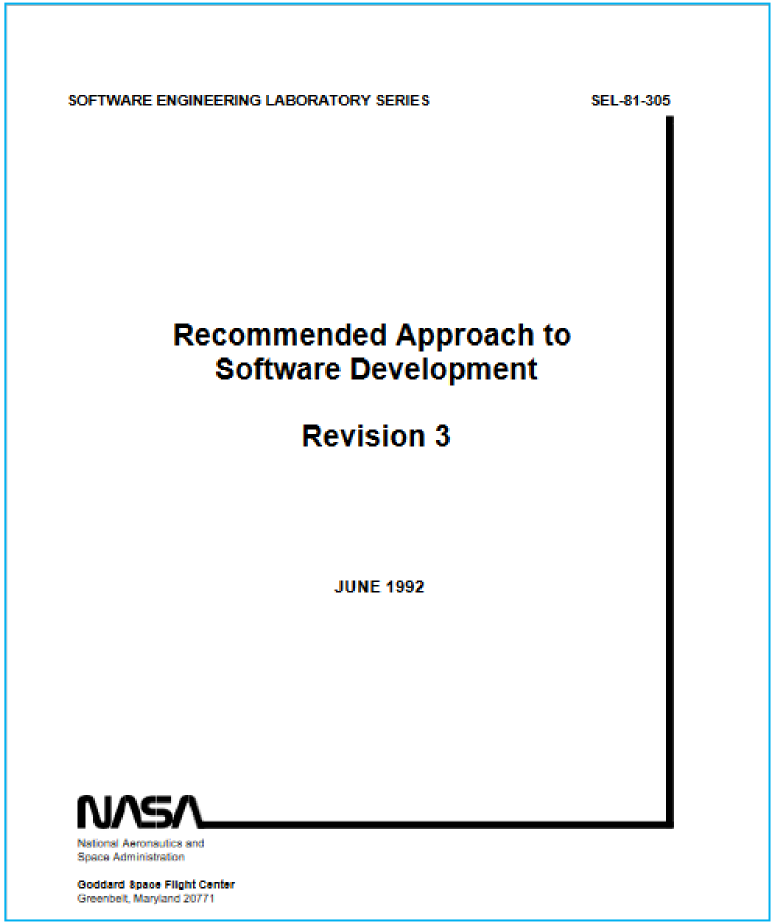
EUROCONTROL @eurocontrol · 1h

A computer failure is affecting London airspace with potentially severe #flightdelay ow.ly/FNheN

289 33

Source: https://www.wired.com/2005/11/historys-worst-software-bugs/

Software Reliability: NASA SEL



SOFTWARE ENGINEERING LABORATORY SERIES SEL-81-305

Recommended Approach to Software Development

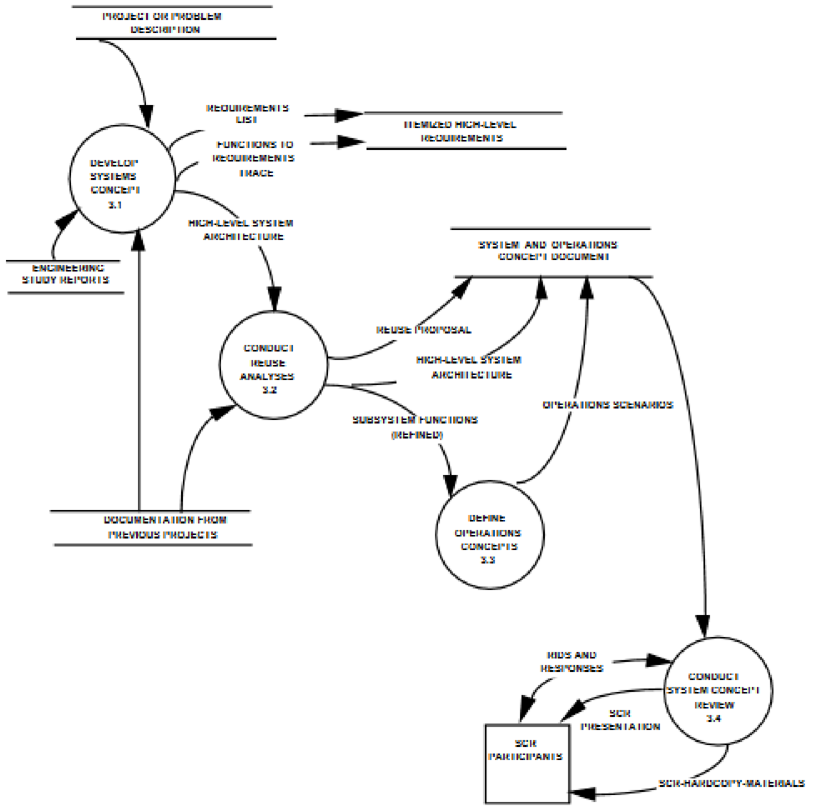
Revision 3

JUNE 1992

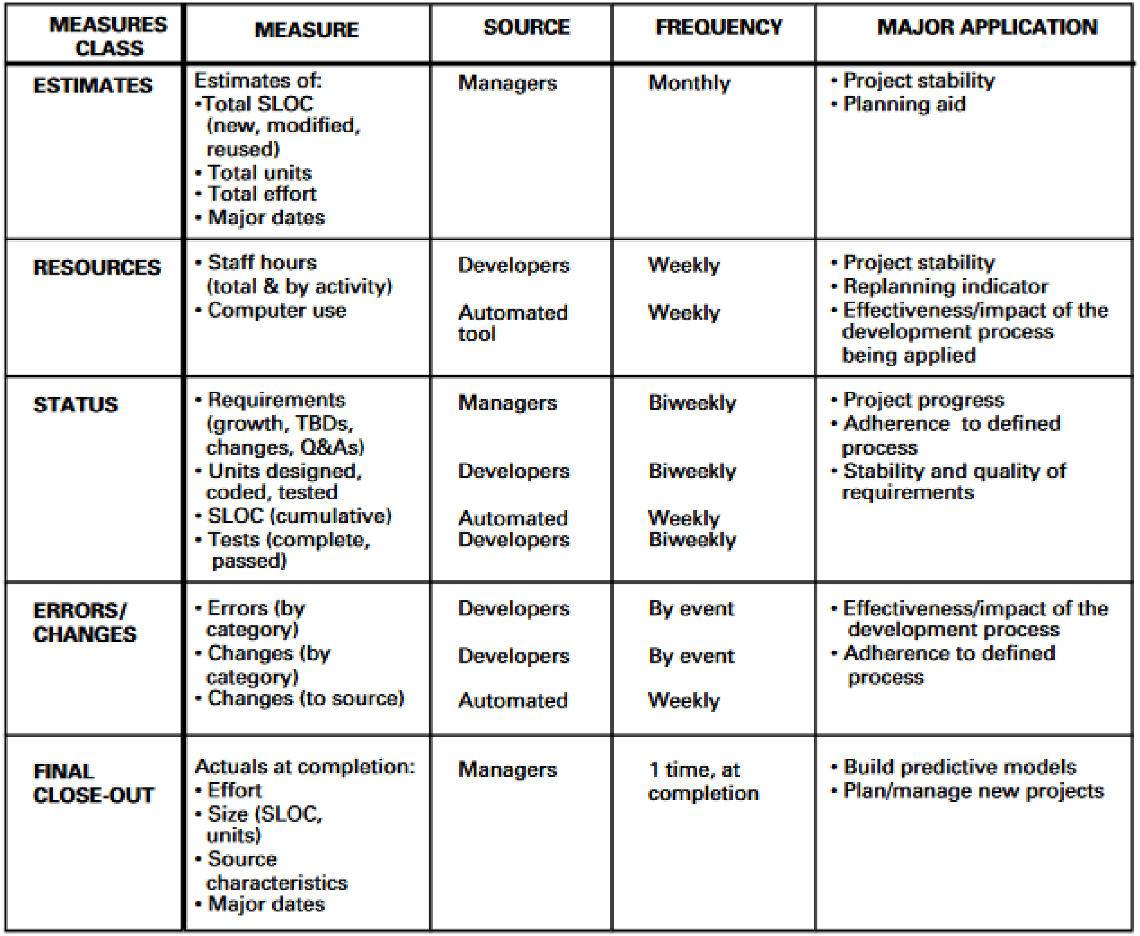
NASA

National Aeronautics and Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

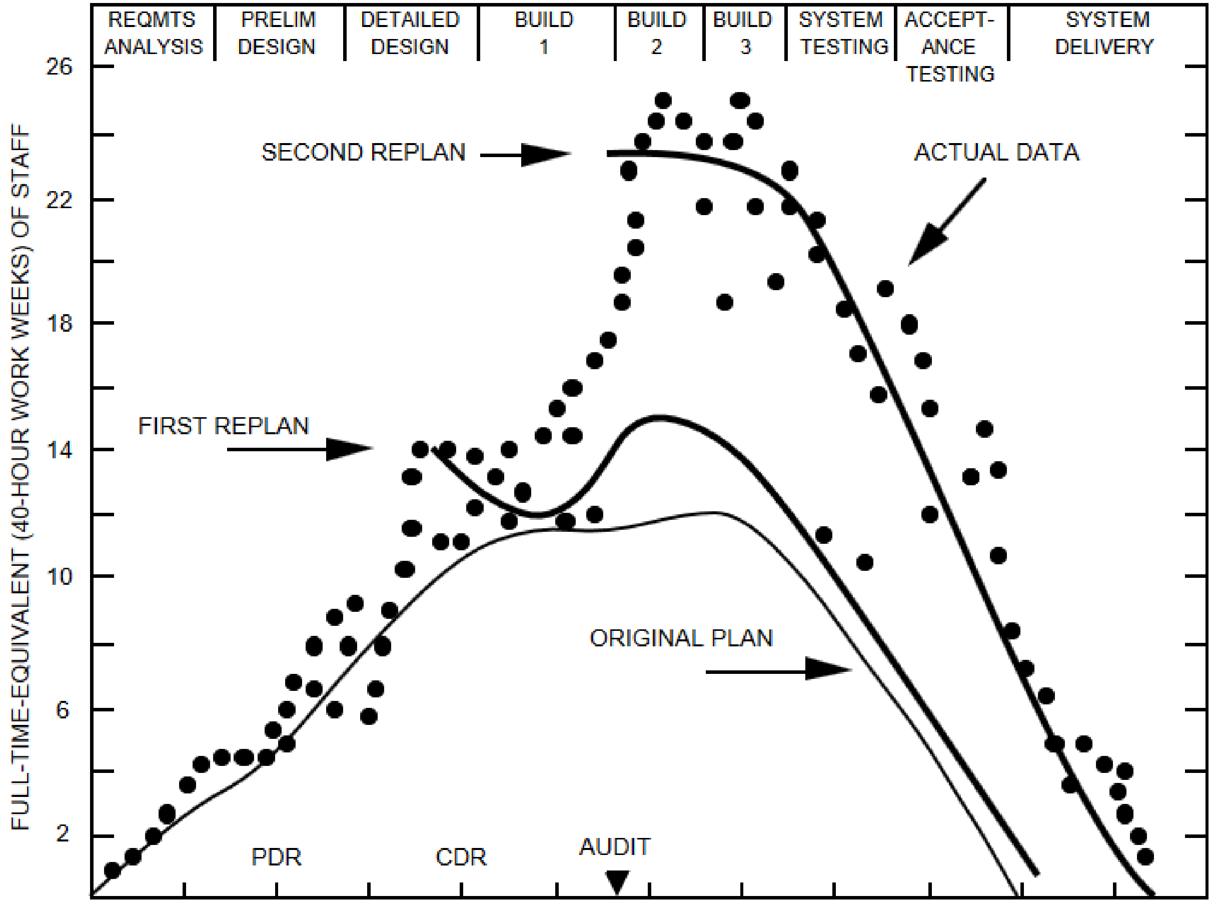

PROJECT OR PROBLEM DESCRIPTION
REQUIREMENTS LIST
ITEMIZED HIGH-LEVEL REQUIREMENTS
FUNCTIONS TO REQUIREMENTS TRACE
DEVELOP SYSTEMS CONCEPT 3.1
HIGH-LEVEL SYSTEM ARCHITECTURE
ENGINEERING STUDY REPORTS
SYSTEM AND OPERATIONS CONCEPT DOCUMENT
CONDUCT REUSE ANALYSES 3.2
REUSE PROPOSAL
HIGH-LEVEL SYSTEM ARCHITECTURE
OPERATIONS SCENARIOS
SUBSYSTEM FUNCTIONS (REFINED)
DOCUMENTATION FROM PREVIOUS PROJECTS
DEFINE OPERATIONS CONCEPTS 3.3
RIDS AND RESPONSES
CONDUCT SYSTEM CONCEPT REVIEW 3.4
SCR PRESENTATION
SCR PARTICIPANTS
SCR-HARDCOPY-MATERIALS

Software Reliability: NASA SEL

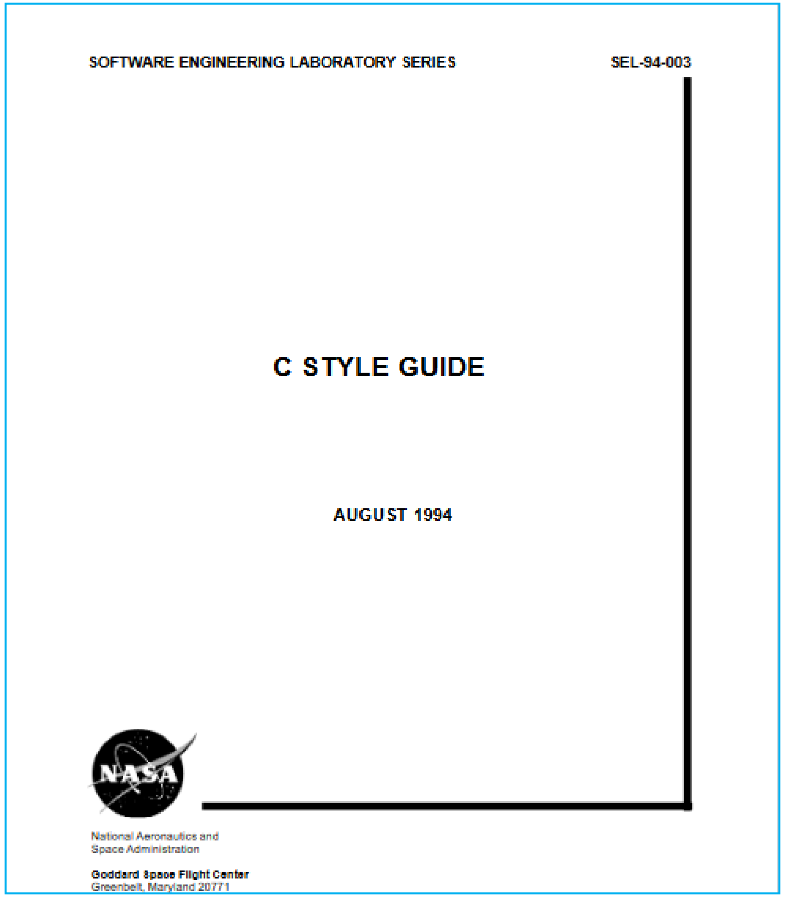


MEASURES CLASS	MEASURE	SOURCE	FREQUENCY	MAJOR APPLICATION
ESTIMATES	Estimates of: •Total SLOC (new, modified, reused) • Total units • Total effort • Major dates	Managers	Monthly	• Project stability • Planning aid
RESOURCES	• Staff hours (total & by activity) • Computer use	Developers Automated tool	Weekly Weekly	• Project stability • Replanning indicator • Effectiveness/impact of the development process being applied
STATUS	• Requirements (growth, TBDs, changes, Q&As) • Units designed, coded, tested • SLOC (cumulative) • Tests (complete, passed)	Managers Developers Automated Developers	Biweekly Biweekly Weekly Biweekly	• Project progress • Adherence to defined process • Stability and quality of requirements
ERRORS/ CHANGES	• Errors (by category) • Changes (by category) • Changes (to source)	Developers Developers Automated	By event By event Weekly	• Effectiveness/impact of the development process • Adherence to defined process
FINAL CLOSE-OUT	Actuals at completion: • Effort • Size (SLOC, units) • Source characteristics • Major dates	Managers	1 time, at completion	• Build predictive models • Plan/manage new projects

Software Reliability: NASA SEL



Software Reliability: NASA SEL



SOFTWARE ENGINEERING LABORATORY SERIES SEL-94-003

C STYLE GUIDE

AUGUST 1994

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

In general, use short comments to document variable definitions and block comments to describe computation processes.

Example: block comment vs. short comment

preferred style:

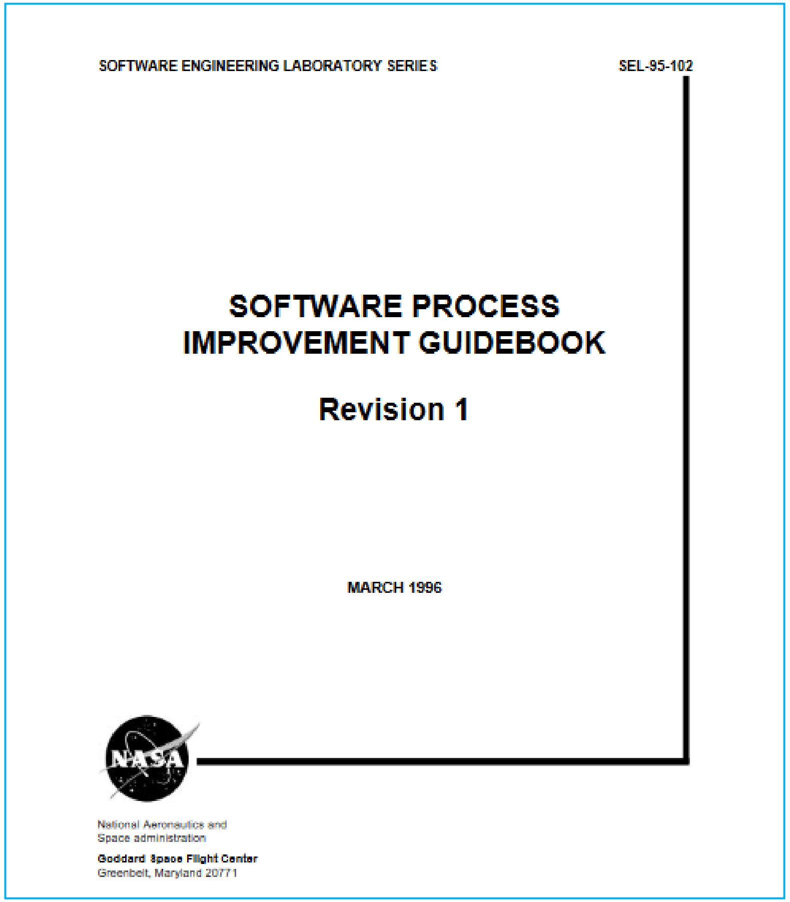
```
/*
 * Main sequence:  get and process all user requests
 */

while (!finish())
{
    inquire();
    process();
}
```

not recommended:

```
while (!finish())  /*  Main sequence:        */
{                  /*                        */
    inquire();     /*  Get user request      */
    process();     /*  And carry it out      */
}                  /*  As long as possible   */
```

Software Reliability: NASA SEL


SOFTWARE ENGINEERING LABORATORY SERIES SEL-95-102

SOFTWARE PROCESS IMPROVEMENT GUIDEBOOK

Revision 1

MARCH 1996

National Aeronautics and Space administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

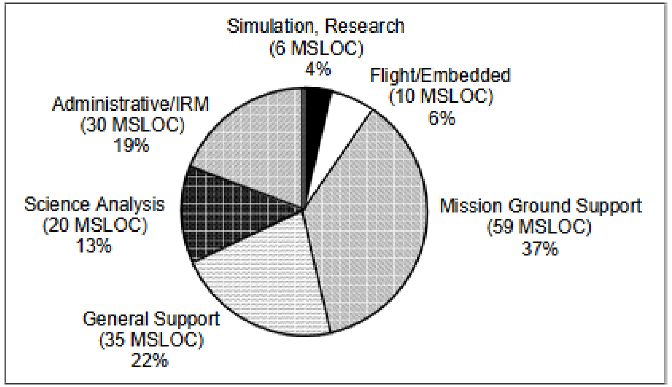


Figure 2-3. NASA Operational Software Domains

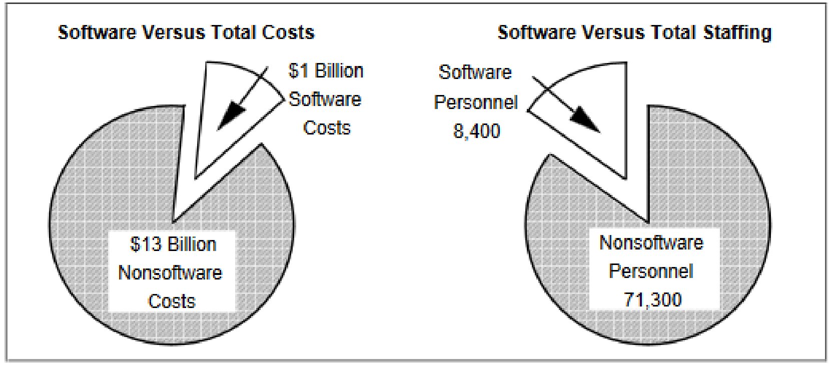


Figure 2-4. NASA Software Resources

Software Reliability: NASA SEL

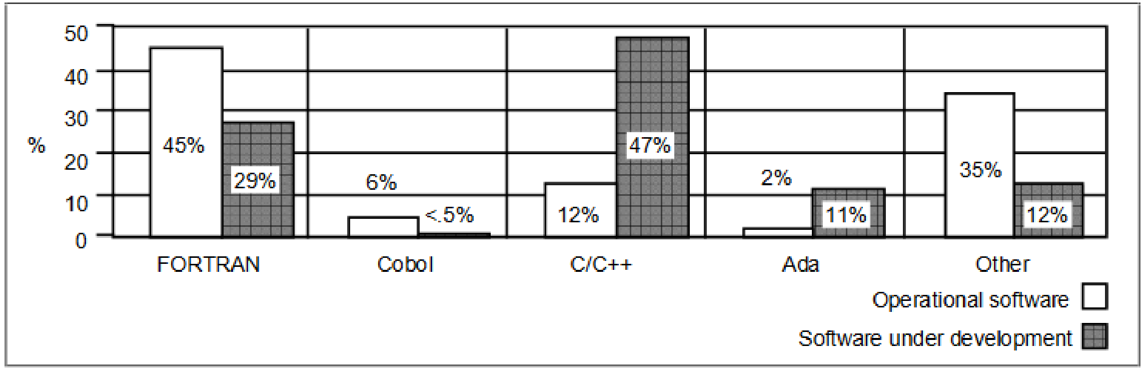


Figure 2-5. NASA Language Preferences and Trends

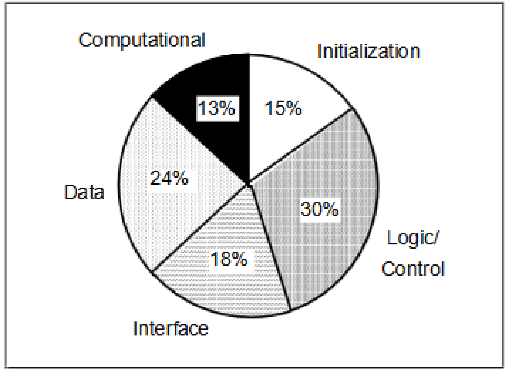


Figure 2-8. Error Distribution by Class

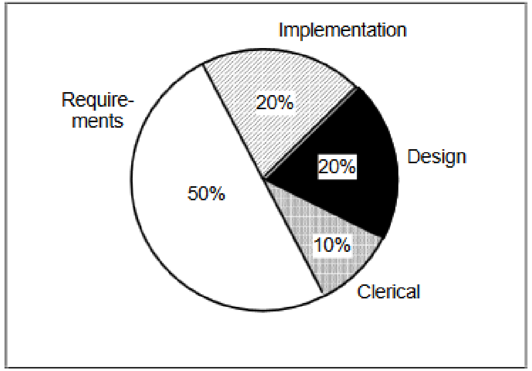
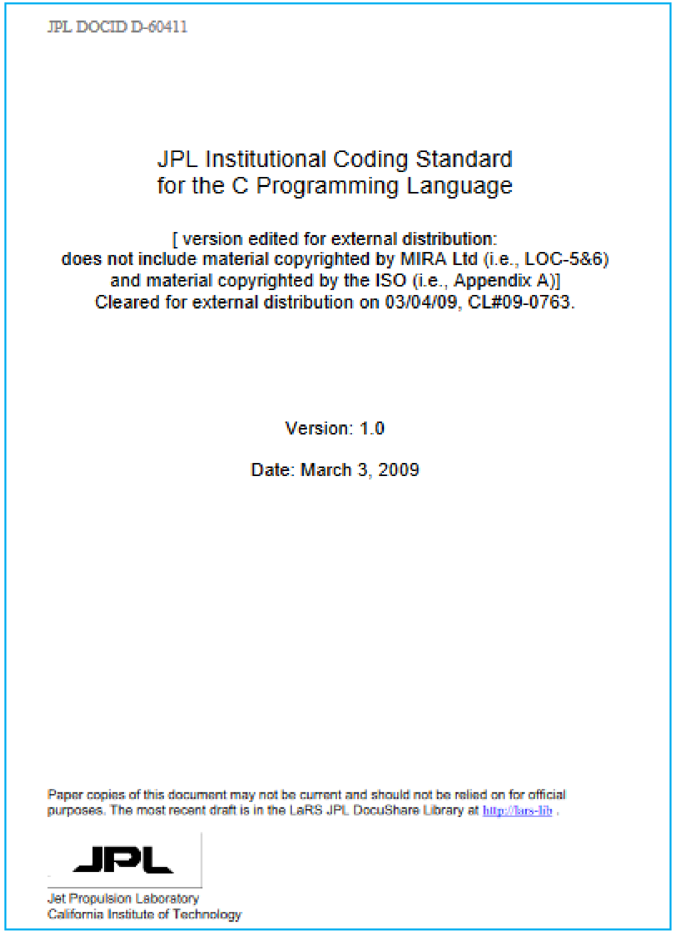


Figure 2-9. Error Distribution by Origin

Software Reliability: NASA SEL



JPL DOCID D-60411

JPL Institutional Coding Standard
for the C Programming Language

[version edited for external distribution:
does not include material copyrighted by MIRA Ltd (i.e., LOC-5&6)
and material copyrighted by the ISO (i.e., Appendix A)]
Cleared for external distribution on 03/04/09, CL#09-0763.

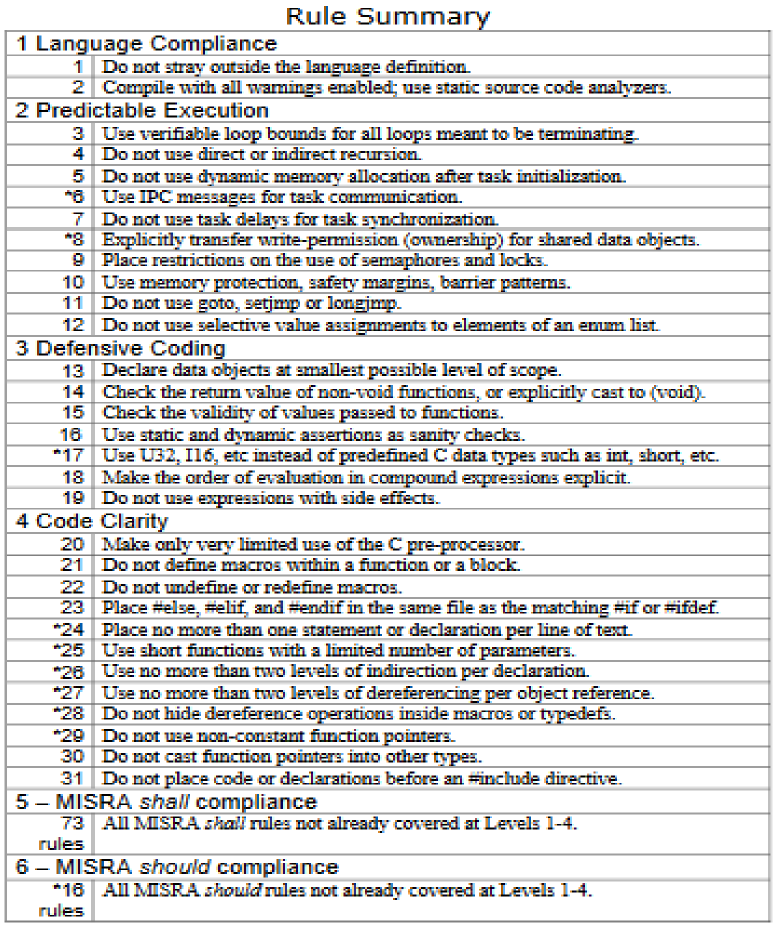
Version: 1.0

Date: March 3, 2009

Paper copies of this document may not be current and should not be relied on for official purposes. The most recent draft is in the LaRS JPL DocuShare Library at http://lars-lib .

JPL

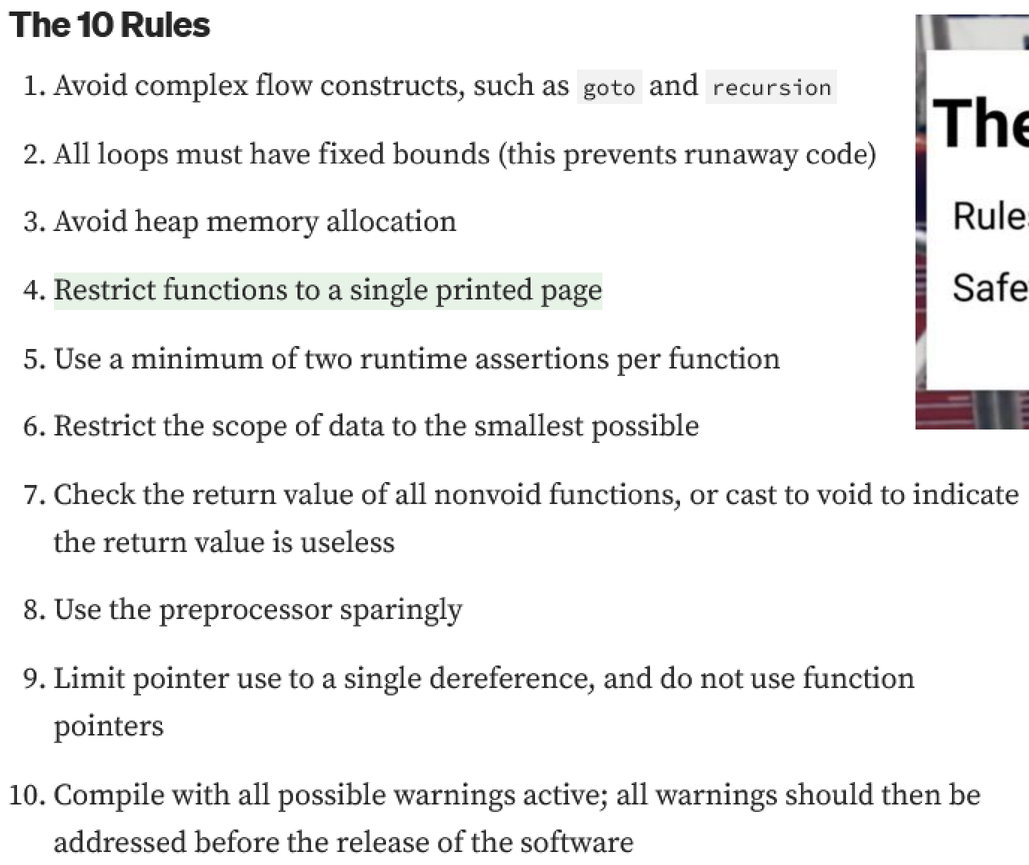
Jet Propulsion Laboratory
California Institute of Technology



Rule Summary

1 Language Compliance	
1	Do not stray outside the language definition.
2	Compile with all warnings enabled; use static source code analyzers.
2 Predictable Execution	
3	Use verifiable loop bounds for all loops meant to be terminating.
4	Do not use direct or indirect recursion.
5	Do not use dynamic memory allocation after task initialization.
*6	Use IPC messages for task communication.
7	Do not use task delays for task synchronization.
*8	Explicitly transfer write-permission (ownership) for shared data objects.
9	Place restrictions on the use of semaphores and locks.
10	Use memory protection, safety margins, barrier patterns.
11	Do not use goto, setjmp or longjmp.
12	Do not use selective value assignments to elements of an enum list.
3 Defensive Coding	
13	Declare data objects at smallest possible level of scope.
14	Check the return value of non-void functions, or explicitly cast to (void).
15	Check the validity of values passed to functions.
16	Use static and dynamic assertions as sanity checks.
*17	Use U32, I16, etc instead of predefined C data types such as int, short, etc.
18	Make the order of evaluation in compound expressions explicit.
19	Do not use expressions with side effects.
4 Code Clarity	
20	Make only very limited use of the C pre-processor.
21	Do not define macros within a function or a block.
22	Do not undefine or redefine macros.
23	Place #else, #elif, and #endif in the same file as the matching #if or #ifdef.
*24	Place no more than one statement or declaration per line of text.
*25	Use short functions with a limited number of parameters.
*26	Use no more than two levels of indirection per declaration.
*27	Use no more than two levels of dereferencing per object reference.
*28	Do not hide dereference operations inside macros or typedefs.
*29	Do not use non-constant function pointers.
30	Do not cast function pointers into other types.
31	Do not place code or declarations before an #include directive.
5 – MISRA *shall* compliance	
73 rules	All MISRA *shall* rules not already covered at Levels 1-4.
6 – MISRA *should* compliance	
*16 rules	All MISRA *should* rules not already covered at Levels 1-4.

Software Reliability: NASA JPL/SEL

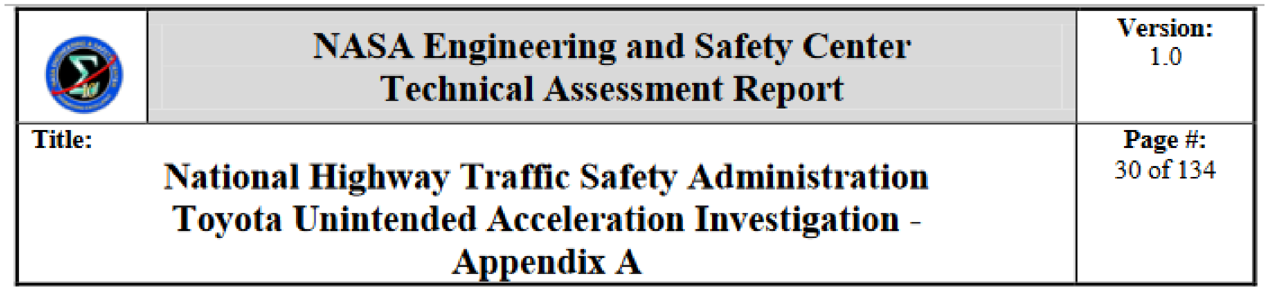


The 10 Rules

1. Avoid complex flow constructs, such as `goto` and `recursion`
2. All loops must have fixed bounds (this prevents runaway code)
3. Avoid heap memory allocation
4. Restrict functions to a single printed page
5. Use a minimum of two runtime assertions per function
6. Restrict the scope of data to the smallest possible
7. Check the return value of all nonvoid functions, or cast to void to indicate the return value is useless
8. Use the preprocessor sparingly
9. Limit pointer use to a single dereference, and do not use function pointers
10. Compile with all possible warnings active; all warnings should then be addressed before the release of the software

Source: https://betterprogramming.pub/the-power-of-10-nasas-rules-for-coding-43ae1764f73d

Software Reliability: NASA JPL/SEL



NASA Engineering and Safety Center Technical Assessment Report	Version: 1.0
Title: National Highway Traffic Safety Administration Toyota Unintended Acceleration Investigation - Appendix A	Page #: 30 of 134

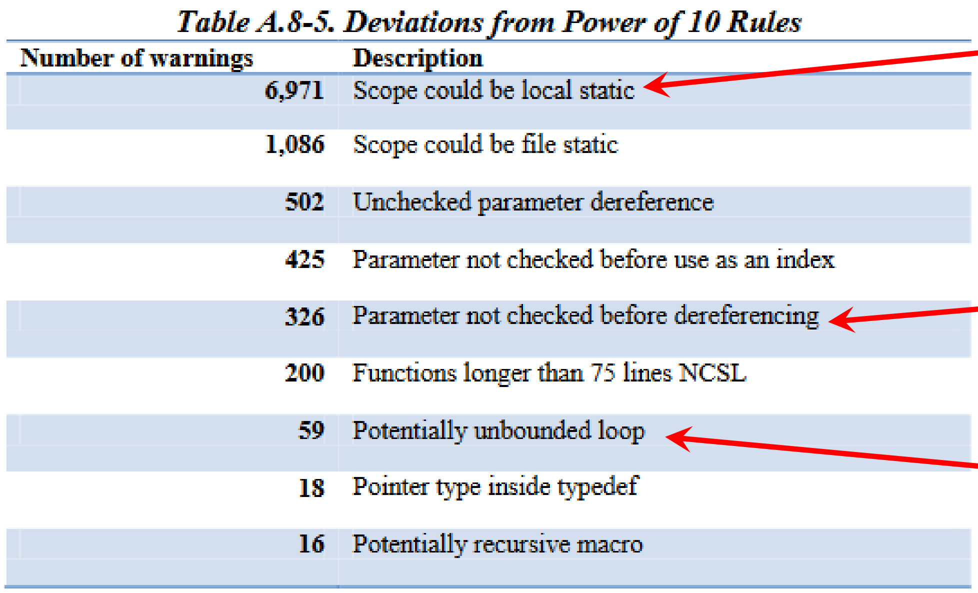


Table A.8-5. Deviations from Power of 10 Rules

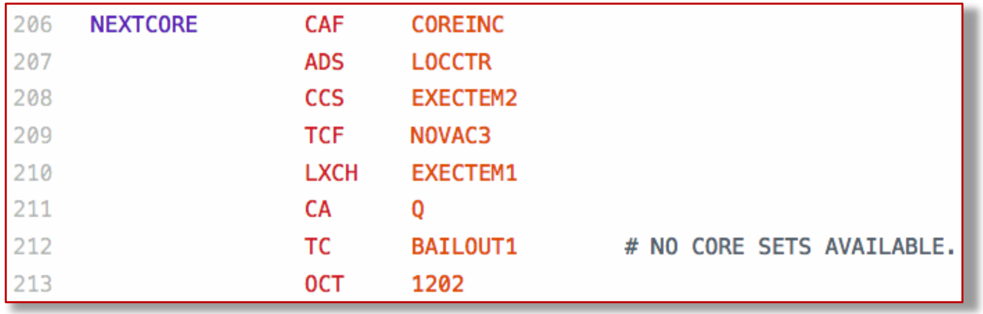
Number of warnings	Description
6,971	Scope could be local static
1,086	Scope could be file static
502	Unchecked parameter dereference
425	Parameter not checked before use as an index
326	Parameter not checked before dereferencing
200	Functions longer than 75 lines NCSL
59	Potentially unbounded loop
18	Pointer type inside typedef
16	Potentially recursive macro

globals...

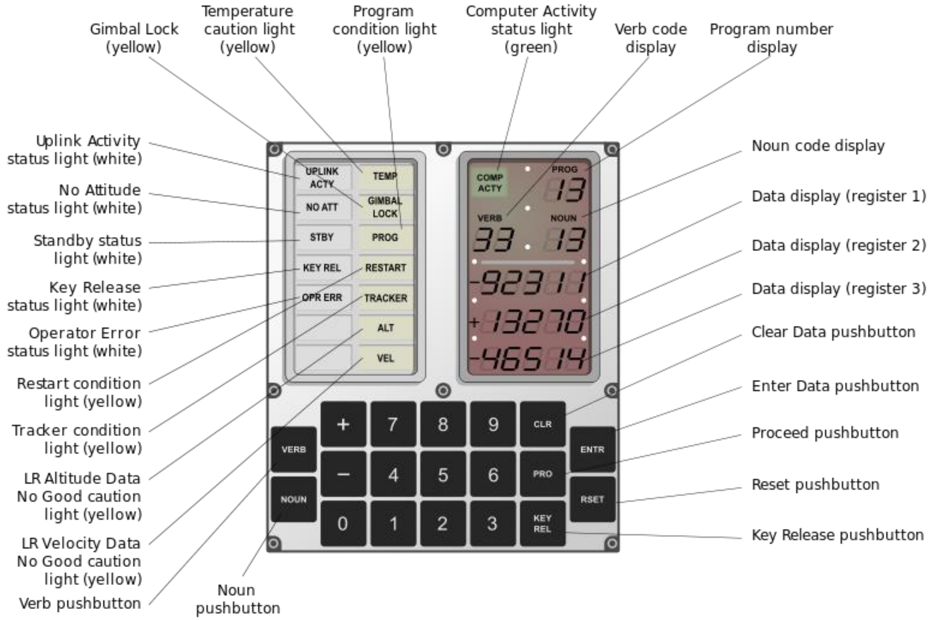
pointers...

hang up...

Hamilton in 1969, standing next to listings of the software she and her MIT team produced for the Apollo project



```
NEXTCORE        CAF     COREINC
                ADS     LOCCTR
                CCS     EXECTEM2
                TCF     NOVAC3
                LXCH    EXECTEM1
                CA      Q
                TC      BAILOUT1        # NO CORE SETS AVAILABLE.
                OCT     1202
```



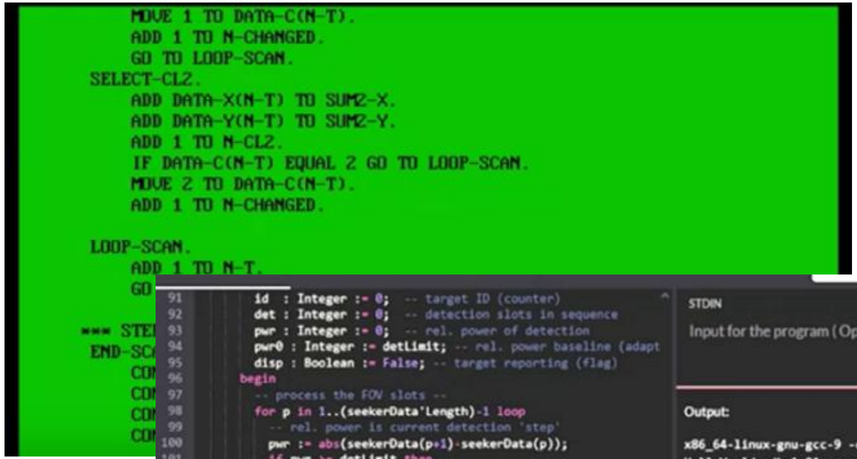
Gimbal Lock (yellow)
Temperature caution light (yellow)
Program condition light (yellow)
Computer Activity status light (green)
Verb code display
Program number display
Uplink Activity status light (white)
No Attitude status light (white)
Standby status light (white)
Key Release status light (white)
Operator Error status light (white)
Restart condition light (yellow)
Tracker condition light (yellow)
LR Altitude Data No Good caution light (yellow)
LR Velocity Data No Good caution light (yellow)
Verb pushbutton
Noun pushbutton
Noun code display
Data display (register 1)
Data display (register 2)
Data display (register 3)
Clear Data pushbutton
Enter Data pushbutton
Proceed pushbutton
Reset pushbutton
Key Release pushbutton
UPLINK ACTY
TEMP
NO ATT
GIMBAL LOCK
STBY
PROG
KEY REL
RESTART
OPR ERR
TRACKER
ALT
VEL
COMP ACTY
PROG
13
VERB
NOUN
33
13
-92311
+13270
-46514
+
7
8
9
CLR
VERB
ENTR
−
4
5
6
PRO
NOUN
RSET
0
1
2
3
KEY REL

See: “The Real Story Behind the Apollo 11 Computer Error” – https://www.youtube.com/watch?v=z4cn93H6sM0

Σύνοψη

- Περιεχόμενα:
  - Τι είναι η «Αξιοπιστία Λογισμικού» και η Διασφάλιση Ποιότητας Λογισμικού (SQA).
  - Γιατί οι μεθοδολογίες διασφάλισης ποιότητας λογισμικού είναι απαραίτητες.
  - Μέθοδοι και μετρικές στις διάφορες φάσεις ανάπτυξης λογισμικού.
  - Μερικά παραδείγματα από τον πραγματικό κόσμο – Διαστημικές αποστολές.

- Πηγές:
  - ISO/IEC 9126 Software Quality Model (1993), ISO/IEC 25010 Software Quality Model (2011).
  - G. Schulmeyer, J. McManus, "Software Quality Handbook", Prentice Hall (1998).
  - IEEE Std 730 (2002), IEEE Standard for Software Quality Assurance Plans, IEEE Computer Society / Software Engineering Standards Committee.
  - Rosenberg, L.H.; Gallo, A.M., Jr., "Software quality assurance engineering at NASA", Proc. IEEE Aerospace Conf. 2002, Vol.5.



```
        MOVE 1 TO DATA-C(N-T).
        ADD 1 TO N-CHANGED.
        GO TO LOOP-SCAN.
    SELECT-CL2.
        ADD DATA-X(N-T) TO SUM2-X.
        ADD DATA-Y(N-T) TO SUM2-Y.
        ADD 1 TO N-CL2.
        IF DATA-C(N-T) EQUAL 2 GO TO LOOP-SCAN.
        MOVE 2 TO DATA-C(N-T).
        ADD 1 TO N-CHANGED.

    LOOP-SCAN.
        ADD 1 TO N-T.
        GO

  *** STE
    END-SC
        CO
        CO
        CO
        CO
```

```
        id  : Integer := 0;  -- target ID (counter)
        det : Integer := 0;  -- detection slots in sequence
        pwr : Integer := 0;  -- rel. power of detection
        pwr0 : Integer := detLimit; -- rel. power baseline (adapt
        disp : Boolean := False; -- target reporting (flag)
      begin
        -- process the FOV slots --
        for p in 1..(seekerData'Length)-1 loop
          -- rel. power is current detection 'step'
          pwr := abs(seekerData(p+1)-seekerData(p));
          if pwr >= detLimit then
            -- detection valid, continue analysis
            if pwr > pwr0+detLimit then
              -- strong new 'step' from baseline (new target)
              pwr0 := pwr;    -- update the baseline
              det :
              disp
            end if;

            det := 
            -- chec
            if (det
              id :=
              -- adj
              displa
              pwr0 
              disp 
```

STDIN

Input for the program (Optional)

Output:

```
x86_64-linux-gnu-gcc-9 -c HelloWorld.adb
HelloWorld.adb:1:01: compilation unit expected
x86_64-linux-gnu-gnatmake-9: "HelloWorld.adb" com
```

- Hamming (7,4) error correction codes in R
- Kmeans clustering in COBOL
- Bi-directional Associative Memory (BAM) in Arduino/C
- Linear Regression in SQL, Matlab
- k-nearest-neighbor Classifier in SQL
- ...

YouTube:

@ApneaCoding

https://www.youtube.com/@apneacoding

Github:

@xgeorgio

https://github.com/xgeorgio

Ερωτήσεις

Χάρης Γεωργίου (MSc,PhD)
https://www.linkedin.com/in/xgeorgio/
https://twitter.com/xgeorgio_gr