Advanced QA training:

Test Analysis techniques and processes

Ondrej Tulach

Senior Quality Assurance Engineer, QA experience since 2010

What does this guy know about QA?

- Former Sysadmin tasked with **hacking*** and debugging obscure Municipal and Education SW and networks
- Former PL/I Mainframe **programmer**, C#.NET developer
- **Experience** with QA since 2010
- Production support & testing → manual testing → automated testing → test analysis → system/safety/reliability analysis
- Experience with QA in Banking, Security, Database systems
- Defence aerospace hobbyist – and **Aerospace** has created QA as SoA
- Experience+Analytics+Aerospace QA+hacking = **better QA for everyone**

*hacking: 1) using for different than intended purpose, deployment; 2) improvized or crafty solution

What you think you know about QA is most likely wrong

„No one ever sits in front of a computer and accidentally compiles a working program, so people know—intuitively and correctly—that programming must be hard.

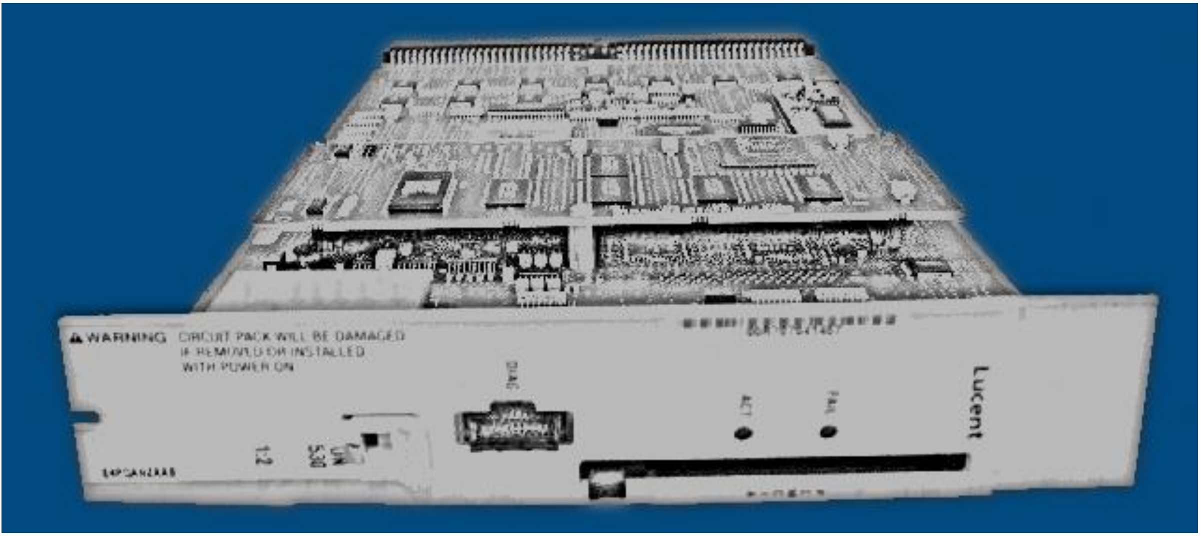
By contrast, almost anyone can sit in front of a computer and stumble over bugs, so people believe—intuitively and incorrectly—that testing must be easy!“

~Michael Bolton, DevelopSense

The „Titanics“ happening without Advanced QA (A.K.A. „we have Unit tests“)

Bell 4ESS, outage of entire NYC AT&T network, 15. january 1990

- SW update for core AT&T switches
- If a switch reboots and sends 1st message, other switches do an internal reset upon receipt
- If other switches received 2nd message while undergoing that internal reset: crash
- Post-crash: automatic restart.
- Avalanching: each restarted sends 1st message, then 2nd… Crashing others
- Recursive fault, all 114 core switches restarting each 6 seconds!
- Whole 9 hours outage
- (Workaround: lower network throughput, downgrade FW)
- **Failure Mode testing && Integration testing**

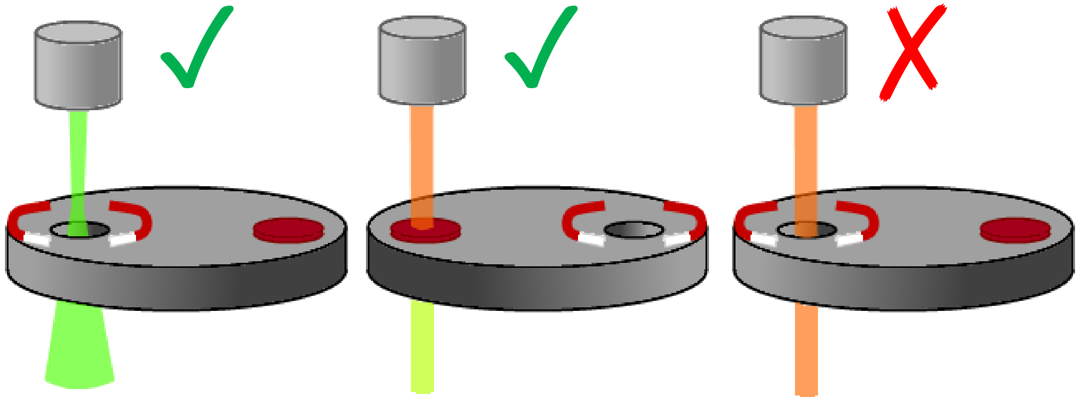


WARNING CIRCUIT PACK WILL BE DAMAGED IF REMOVED OR INSTALLED WITH POWER ON
DIAG
ACT
FAIL
Lucent

http://phworld.org/history/attcrash.htm

Therac-25, iradiation of patients, 1985-1987 (3 dead, more radiation sickness)

- Radiotherapy machine had 3 opmodes:
  1. Low-energy iradiation with magnetically deflected ray
  2. Indirect iradiation of 100× higher energy with mask inserted (for conversion to RTG radiation)
  3. Light beam for aiming
- Therac-25 has replaced mechanical safeties with SW
- Race conditions were not handled
  - If operator selected mode 2) and fast-enough switched to mode 1), the 100x power iradiation of was selected yet the shield was removed
  - If operator selected mode 3) and fast-enough switched to mode 1) it irradiated one spot without EM deflection, causing deep burns
- **Static inspection; State transition testing**



http://radonc.wikidot.com/radiation-accident-therac25

Ariane 5G, flight 501 explosion, 4th june 1996 (370 milion USD loss)

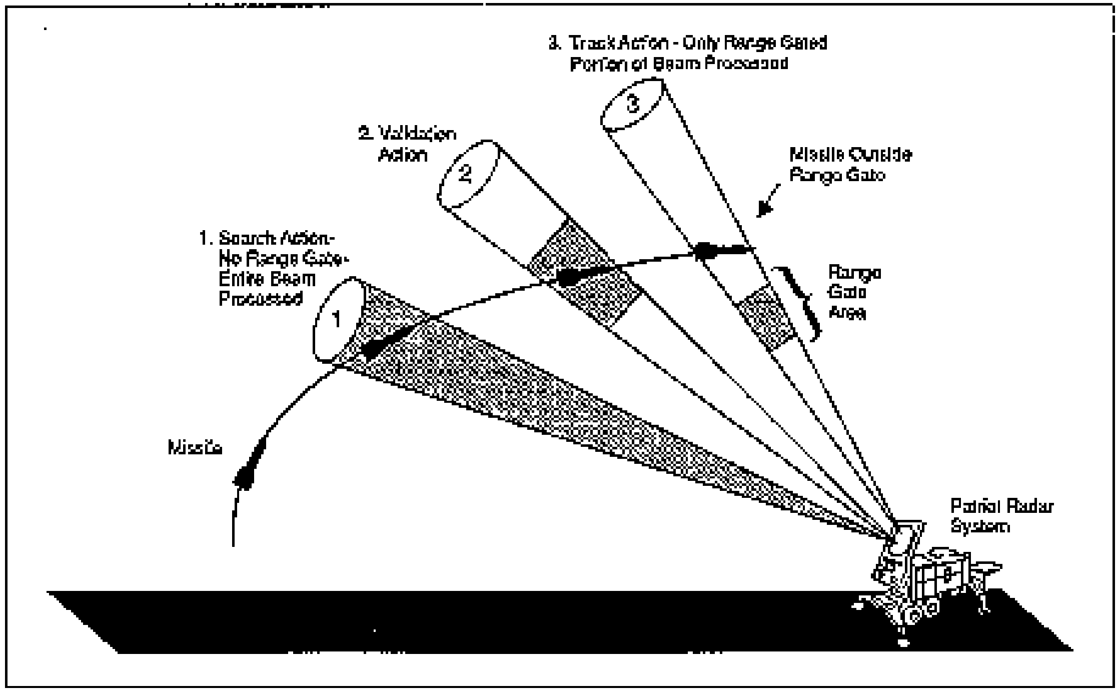
- Defect: converting from 64b float to 16b sint; Ariane5 values did not fit into 16b → overflow
- ADA language detects overflows, raises exception
- Error: code threw OPERAND_ERROR, SRI dumped memoryon output
- Failure: the dump was interpreted as flight data!
- System interpreted the dump „flight data“ as extreme path deviation; „corrected“ by full control surfaces deflection and 20° turn
- Parallel PRI+BKP failure – hot backup, same toxic data
- Aerodynamic forces broke the rocket in half
- And the code was useless dead-weight!
- **FTA with CCF, Stress testing, Integ.T with FMT, Boundary Values Testing A5G**

http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html

MIM-104 Patriot, BMD failure, 25th feb 1991 (28 dead, 100 wounded)

- Patriot SW fault prevented shootdown of SCUD attacking missile
- Patriot radar lost tracking due to faulty calculation of distance to look for the target
- The radar distance-to-look calculation was function of
  - Target velocity
  - Patriot systém clock (since system start)
- Defect: rounding error added offset 0,000000095d every tick (1/10 sek.)
- Consequence: radar distance-to-look value shifted +7 meters each hour; LoM after 20 hrs
- The failing Patriot BMD was uptime for 100 hours, missed SCID, SCUD hit barracks
- **Stress testing, Boundary values testing… <u>Challenging DEV assumptions!</u>**

Figure 5: Incorrectly Calculated Range Gate


3. Track Action - Only Range Gated Portion of Beam Processed
3
2. Validation Action
2
Missile Outside Range Gate
1. Search Action- No Range Gate- Entire Beam Processed
1
Range Gate Area
Missile
Patriot Radar System

GAO/IMTEC-92-26, used under fair-use

http://www-users.math.umn.edu/~arnold/disasters/patriot.html

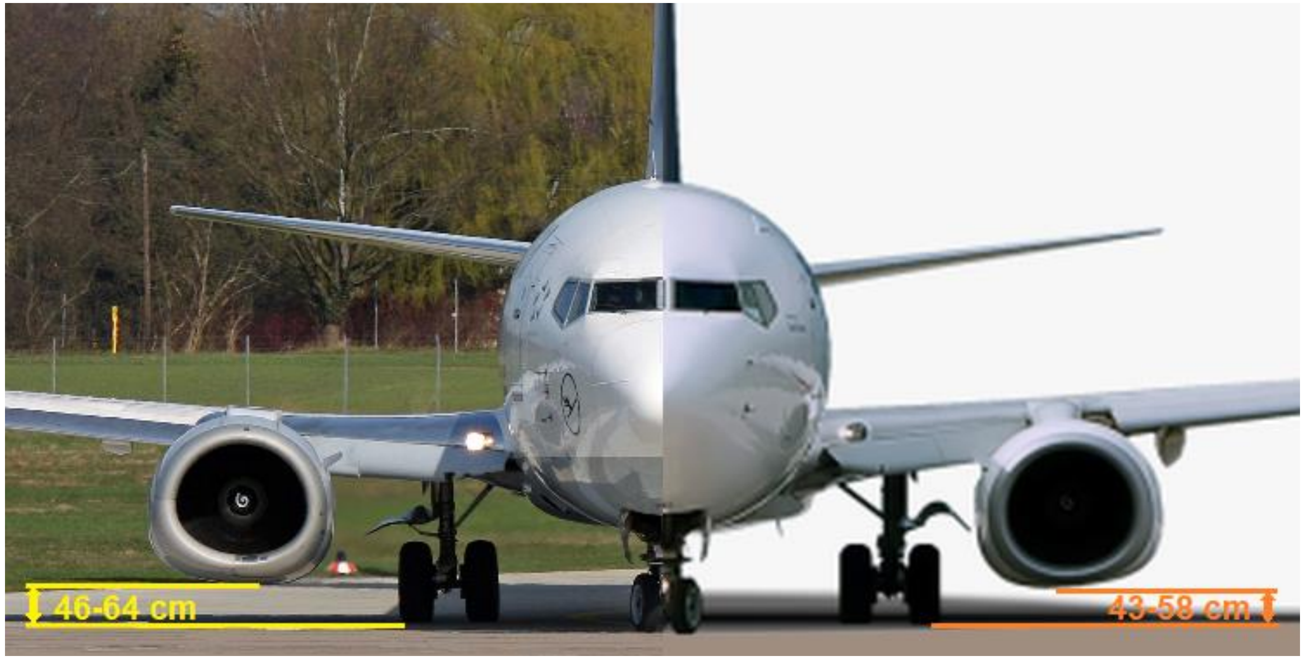
Airbus 330-203, stall and LOV, 1th June 2009 (all 228 souls onboard dead)

- Aircrew stalled aircraft, crashed into sea
- Complementary causes: dangerous SW implementation of crew alerts
- Issues between Design and SW implementation
  1. **FBW changes signalling**
     a) FBW in Normal Law (I) prevents Stall pilot input
     b) FBW in Alternate Law (II) allows Stall pilot input
     c) Transition between FBW laws (I) into (II) signalled by 1 beep, little LCD cross, little text outside field of view
  2. **Reversed „Stall“ alarm 🔈 suppression**
     a) Stall alarm suppressed in low airspeed
     b) Intention: during landing – but implicitly also during extreme stall!
     c) If pilot attempts recovery, airspeed increases → alarm suppression deactivates → stall alarm activates **negative feedback**
- **FMEA and FTA of „Stall alarm“ subsystem false positive and false negative**

https://abcnews.go.com/International/air-france-flight-447-investigation-pilots-properly-trained/story?id=16503005

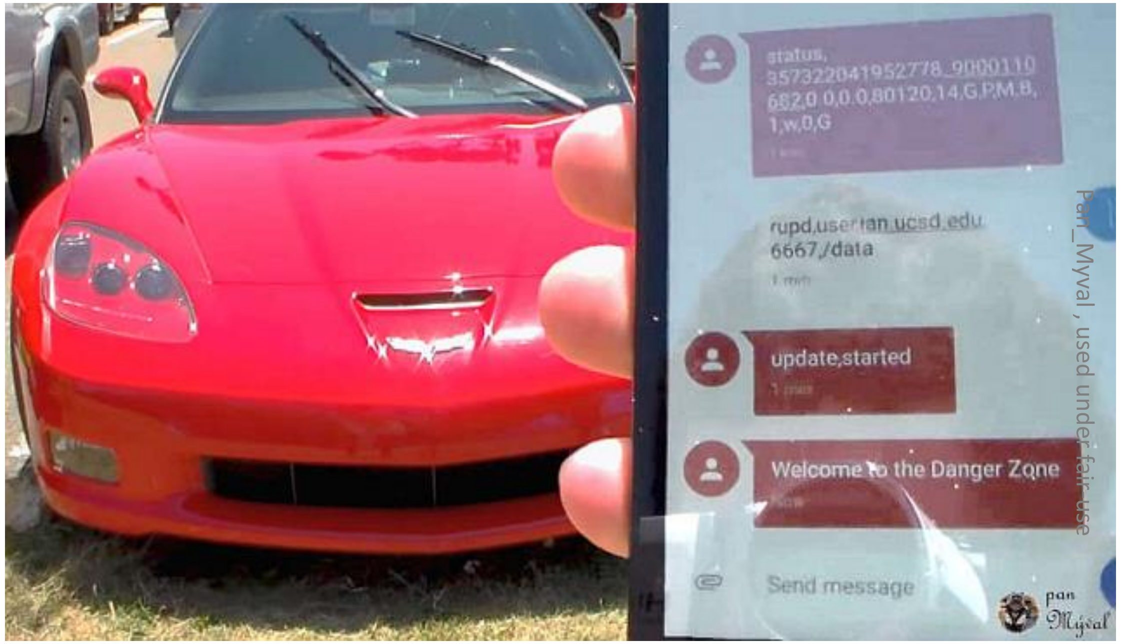
Boeing 737 MAX crashes

- New engines to compete with A320 Neo
- Engines too large to fit normally -> CG shift
- Flight handling change...
- Re-cert or sim training → $1M discount
- MCAS to „hide“ changed handling
- Only single AoA sensor → SPoF
- QA certified as non-full-authority
- MGMT changed to full authority, but no re-analysis!
- Boeing QA pushed to not report risks and defects
- Documentation of known failure of „AoA disagree alert“ SW was knowingly deferred for 3 years!
- 346 people died in 2 crashes



Multiple CANbus-equipped cars, critical function hacker takeover, 2015

- CANbus is the „spine“ of modern cars
- CAN commands can eg. Deactivate brakes via ABS
- Parking/Lane assist obeys CAN commands
- Any „online“ device connected to CANbus is an attack vector and vulnerability
- Jeep Cherokee MY14 - Infotainment vulnerable *and* interconnected with car‘s CANbus:
  - Complete brake deactivation within low speeds
  - Remote switch to „Parking assist“ mode on highway
- Insurance company remote telemetry OBD dongle
  - Chevy Corvette MY14, Ford Escape MY13, Toyota Prius – takeover of wipers, brake deactivation
- **Input sanitization validation including Authorization testing… Function list to ID auth neccesity!**



status, 357322041952778_9000110 682,0 0,0.0,80120,14,G,P,M,8, 1,w,0,G

rupd,user[illegible]an.ucsd.edu 6667,/data

1 min

update,started

Welcome to the Danger Zone

Pan_Myval , used under fair-use

Wired: https://youtu.be/MK0SrxBC1xs

IHS Markit – Security and the connected car

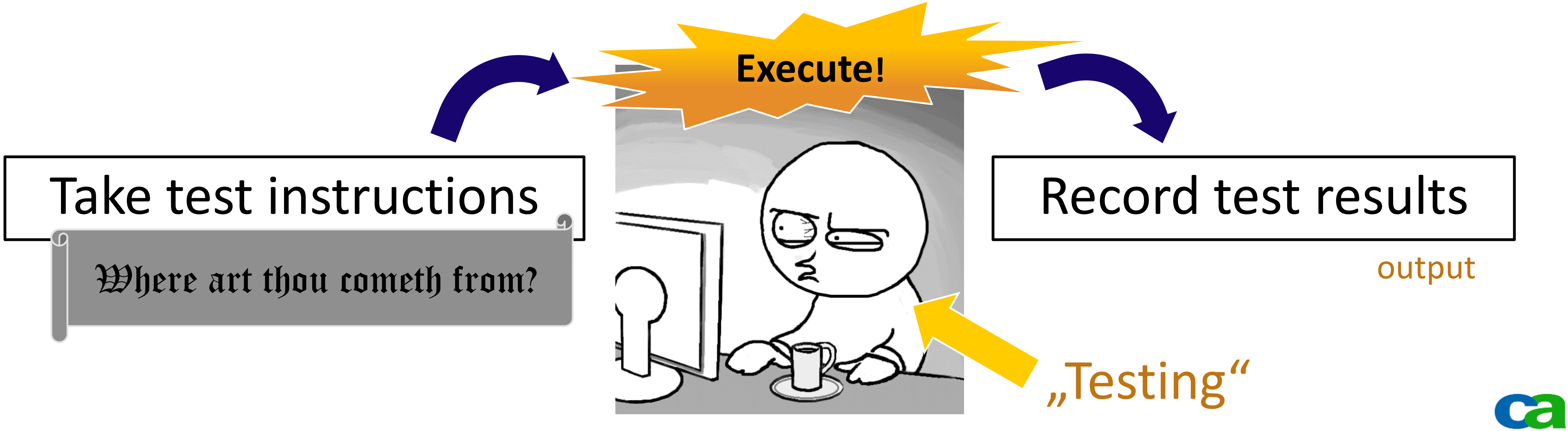
https://www.autickar.cz/blog/clanek/odborna-prednaska-hackovani-a-ovladnuti-modernich-aut/529/

Quality Control, Quality Assurance, Test Analysis

What they are, what they aren‘t, and why it matters

Testing is just the tip of an iceberg

- *„Procedure of submitting a subject of test to such conditions or operations as will lead to its proof or disproof or to its acceptance or rejection”* (Merriam-Webster)
  - Generalization: intent is logically separated from testing as activity. Decompose more...
- *„Subjecting a system to certain activities and recording system response“*
  - This is the core definition of testing. Everything else is connected but logically separated activity



Take test instructions

Where art thou cometh from?

Execute!

Record test results

output

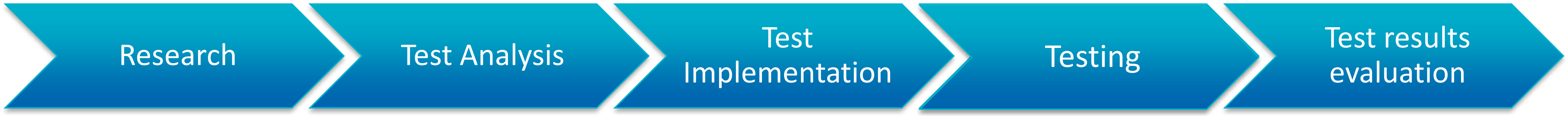
„Testing“

Testing is „Quality Control“. Yet Quality Assurance is far more

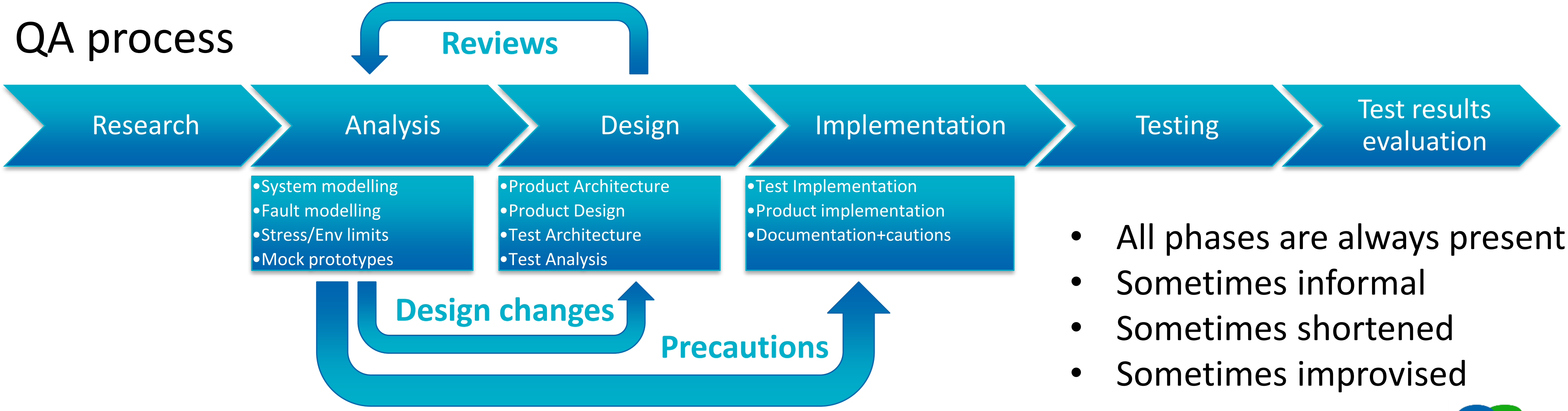
- Quality Control vs. Assurance
  - QC = **discover** defects in finished product not to ship them to customer
    - Person at the end of assembly line throwing defective products aside so they‘re not shipped
  - QA = **prevent** defects during and before product creation
    - Person who tunes the assembly line robot to produce fewer defective products
    - Person who designs the assembly line so that it will be resilient to robot defects
    - Person who designs the product to work despite assembly robot errors
    - And also QC because no preventive effort is ever 100%
- QC is about
  1. testing procedures (measuring, trying...), period.
- QA is about
  1. analysis (identify what could fail),
  2. processes (prevent the defect from being made)
  3. robust design (overcome/suppress defects)
  4. better QC (identifying what to check even when not obvious)

Always present workflow: QC and QA

QC process



QA process



- System modelling
- Fault modelling
- Stress/Env limits
- Mock prototypes

- Product Architecture
- Product Design
- Test Architecture
- Test Analysis

- Test Implementation
- Product implementation
- Documentation+cautions

- All phases are always present
- Sometimes informal
- Sometimes shortened
- Sometimes improvised

Who is QA engineer? Any of these. Often all-in-one

- QA Engineer is broad term covering all preventive professions:
  1. **Tester**: executes the test cases, verifies results and reports discrepancies
  2. **Test Analyst**: designs test cases through systematic analysis and specific techniques to discover possible issues, bugs, dangerous scenarios etc.; provides feedback ad design to developers
  3. **Architect**: supervises good design, maintenability, interoperability, integration, growth-proof, weak points, vulnerabilities etc., mostly via reviews
  4. **Test Architect:** choses appropriate testing strategy, test analysis techniques
  5. **Safety Analyst:** uses expert methods to discover otherwise unnoticed or unexpected failure modes and nodes
  6. **Test Automator**: makes the test cases automated to avoid manual testing

Verification, Validation, and Quality

Static + Unit + Integration

- Verification: implementation – *„Are we building the product correctly?“*
  - Is the product, system, subsystem... properly implemented?
  - Is the product, as designed, consistent, defect-free and sanitized?
  - Is the product built well, without hacks, dead code, race conditions...?

System + Acceptance

- Validation: design – *„Are we building the correct system?“*
  - Did engineers/developers understand the requirements correctly?
  - Are the implicit design assumptions really substantiated and correct?
  - Could the customer use the product for intended purposes?

Fault analysis + System + Stress + Environmental

- Quality: other aspects, unintended consequences
  - Is the design enough robust, maintenable, servicable, easy to use?
  - Even if the product is built to-specs, and customer is happy with it, could it cause problems?
  - Could the system catastrophically fail or cause damages or harm
    - When encountering unexpected, yet realistically possible environment?
    - When run to it‘s limits?
    - When it transitions through unexpected sequence of states?
    - When some of it‘s dependencies or elements unexpectedly failed?

Test analysis effort and skillset needed

QA Maturity levels: what is considered in TA and who could do it?

Testers or anyone:
Devs, managers…

1. **Level 1: positive tests only (QC) – 100 % code coverage**
   - Focus on proving correctness: *„I will demonstrate that the SW works as advertised“*
   - Assumption: only correct, expected inputs are provided: *„demo that the precise way I use it, SW works“*
   - No quality guarantee: if you deviate from the demo scenario, anything could happen

QAE = Tester +
Test Analyst

2. **Level 2: destructive tests (QC)**
   - Focus on destruction: „how can I break it“? → find defects in-dev, not in-the-wild
   - Challenge developer‘s assumptions, discover un-sanitized inputs
   - Systematically pursue top achievable code, but still implementation-focused
3. **Level 3: reduce risks (QC+QA)**
   - System-focused: reduce overall risks even due to external influences; review design
   - QA engages in tests – but also in design, architecture etc., to warn about possible dangers early = cheaply
   - Environmnental and integration testing, „testing of tests“ (=processes, coverage checks – eg. Test Classes...)

Expert QA
analysts

4. **Level 4: preventive actions accross entire team/company (QC+QA)**
   - All of the above, plus systematic expert QA models for defect prediction, tools and processes, reviews...

Test analysis effort and skillset needed

Level 4 QA can be done without bureocracy and even in Agile

- When we adopted formal „Test Analysis“ task/phase, TCs increased ca. 100 %
  (From sometimes as low as 20 to average 50 scenarios / 250 Test Cases per single feature)*
- When we adopted „Test Classes“ technique, TC increased further ca. 100 %
  (To average 80~100 scenarios / 500 Test Cases per single new feature)*
- „Quality PI“ gave us chance to implement techniques straight from NASA handbook (Defect RCA+FTA) to further double increase critical „pain feature“ coverage
  (From 100 to 200 scenarios / 1,000 Test Cases per single feature, plus vastly increased QA/SE confidence)*
- Since 2016, all tests for new features are automated and TestDataGen provides fast recreation test data for almost any issue
  (Automation was 1st, Test Analysis introduced right after basic Automation was usable)

*each scenario is run in several modifications with various Db2 variables and ENV combinations, hence in average: Test Cases = 5~7 × Scenarios

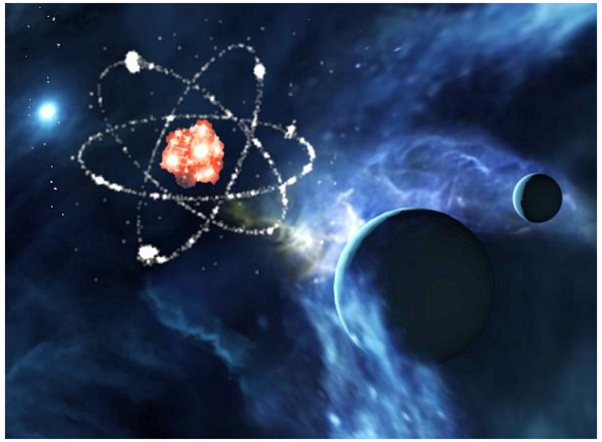
QA Maturity Levels: techniques and approaches employed

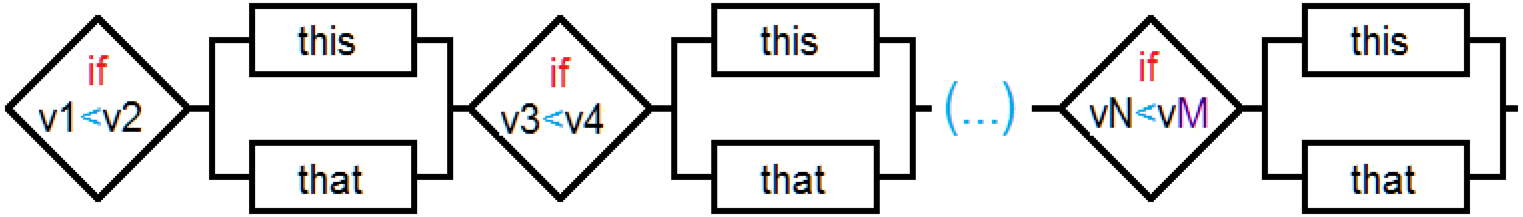
Anyone (test technicians) | Test Analyst + test technicians | TA+TT+Expert analysts

DEBUGGING	VERIFICATION TESTING	DESTRUCTIVE TESTING	RISK REDUCTION	PREVENTIVE QA
Ad-hoc tests	Prove SUT works with: ▪expected inputs ▪customer use-cases ▪industry standards	All from L1, +	All from L1 + L2 +	All from L1 + L2 + L3 +
No reproducibility		BVTs, EPs+DIs	QA of QA: Test Classes	Formal iterative loops design↔analysis
Pass: once ¬crash		HOET	Architects, Test Analysts	
	▪Prove negative function: ▪correct error messages ▪no false positives	FMT	Ad-hoc Design feedback	Models to find defects: ▪State Transition Testing ▪Fault Tree Analysis ▪FMEA ▪Data-flow testing
		Stress testing	Environmental testing	
	Prove client compatibility	Load testing	Fault Insertion Testing	
			Regression testing	
			Function Lists	Escaped defects RCA

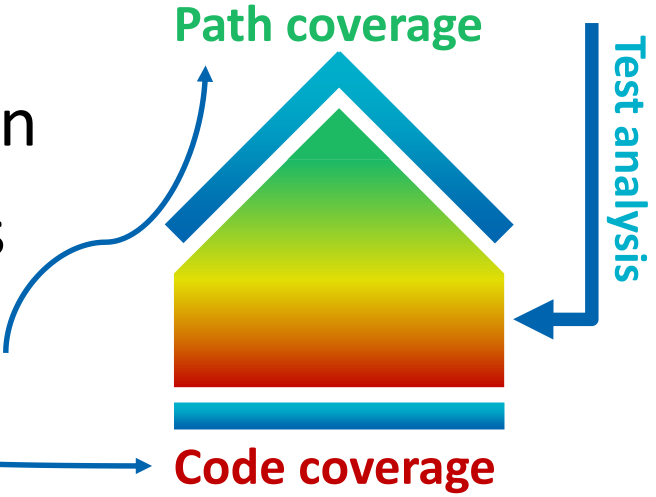
Test analysis effort and skillset needed

The „Test Coverage house“: why automated brute-force testing isn‘t solution



- 100% **Code coverage**: execute each statement
- 100% **Path coverage**: test all paths thru branches
- Example: code coverage=$2\times m$, path coverage = 2^m TCs
- 

- Simple program with 70 branches: $2^{70} \cong$ 1,2 sextillion
- Automated testing with 30sec á test case: 10^{15} years
- 100% Path coverage = maximum but rarely possible
- 100% Code coverage = reasonable minimum
- Test analysis = hand-pick important test cases



It‘s worse. Even 100% *Path* coverage assures nothing – because Data Flows

- Imagine such simple code (just 2 branches)

```
private static float DivNumbers(float a, float b)
{
    if(a != 0 && b != 0)
        return a / b;
    else
        return 0;
}
```

- And we build 100% automated tests:

```
private static void TestIt(float[,] testTriplets)
{
    float currRes;
    for(int fa=0; fa<testTriplets.GetLength(0); ++fa)
    {
        try
        {
            currRes = DivNumbers(testTriplets[fa,0], testTriplets[fa,1]);
        }
        catch(Exception ex)
        {
            Console.WriteLine("ERROR in Test {0}: {1}",fa, ex.Message);
            currRes = float.NaN;
        }
        Console.WriteLine("Test {4} "+
            (float.Equals(currRes, testTriplets[fa,2])?"PASSED:\t":"FAILED: \t")+
            "{0} / {1} => {2} returned vs. {3} expected",
                testTriplets[fa,0], testTriplets[fa,1], currRes, testTriplets[fa, 2],
                (fa<10?"0"+fa.ToString():fa.ToString())
        );
    }
}
```

- With just these 2 test data sets:

```
float[,] testTriplets = new float[,]
{
    //dividend operand, divisor operand, expected result
    {1.0f, 1.0f, 1.0f}          //code path 1
   ,{0.0f, 0, float.NaN}        //code path 2
};
```

We achieved 100% automation, 100% code coverage, 100% path coverage!

- Enough? Really? What about other tests?

```
,{10.0f, 2.5f, 4}            //larger div smaller
,{10.0f, -5.0f, -2.0f}       //negative divisor
,{1.5e-45f, 1, 1.5e-45f}     //BVT min pass
,{1f, 1.5e-45f, float.PositiveInfinity} //BVT 1:min
,{1, 3.4e38f, 2.941177e-39f}//BVT 1:max
,{0.0f, -0, float.NaN}       //BVT negative zero
,{float.PositiveInfinity, float.PositiveInfinity, float.NaN}
,{1.0f, 3.0f, 0.3333333333333333333333f} //periodic
,{4195835.0f, 3145727.0f,
    1.333820449136241002 5f} //Pentium FDIV bug
```

- Point: not just the code flows…
  But also the data flowing through!

technologies

QA mindset
Divergent VS convergent thinking

QA‘s mindset

- *Reviewer* and *Test Analyst* roles are the core of QA‘s job and personality
- Differences between QA and developers: mindset and way of thinking
  - Developers (problem solvers):
    - convergent thinking
    - „how-to-achieve“ mindset
  - Test analysts (problem finders):
    - divergent thinking
    - „what-if“ mindset taken to the extreme
- Additionally, ISTQB states that QA should be:
  - **curious**, professionally **pessimistic**, with **critical** eye, attentive to **details**
  - that doesn‘t mean being insulting and critical about developer‘s work! QA‘s mission is to help the developer, not to be the Grim Reaper!

Convergent VS Divergent thinking

- Coined by Joy P. Guilford, popularized by Malcom Gladwell
- Convergent thinking: many inputs → few outputs
  - Finding „one correct solution“, eg. solving math equation with many variables
- Divergent thinking: few inputs → many outputs
  - Identifying many alternative solutions, original uses of things, problems not seen by others – eg. MacGyver, inventors
- The job role problem:
  - **Developers (& Test Automation) must use Convergent thinking**: they get requirements and must devise and code single correctly working solution
  - **Test Analysts must use Divergent thinking**: they get single solution (or specs) and must find all the things which could go wrong; everything the Convergently thinking Developers couldn‘t see, have ignored for the sake of wrong assumptions or haven‘t even thought could happen
  - Both Convergent & Divergent thinking are critically needed, go hand-in-hand!

Let‘s play a game

- Guilford‘s Divergent Thinking Fluence&Originality assessment
  - *„Find original uses for common items. Be creative. Be original.“*
  - See which and how many uses you could find for a

    normal Brick
  - Test Analyst needs to write down both ordinary, apparent and boring as well as original, exciting and improbable uses – so write down both to get points!
  - Scoring:
    - **3 points** if your listed use of the item is unique within the whole group
    - **2 points** if your listed use of the item is shared by max. 2 other people but not the whole group
    - **1 point** if your listed use of the item is shared by more tthan 1 person

Why is this important?

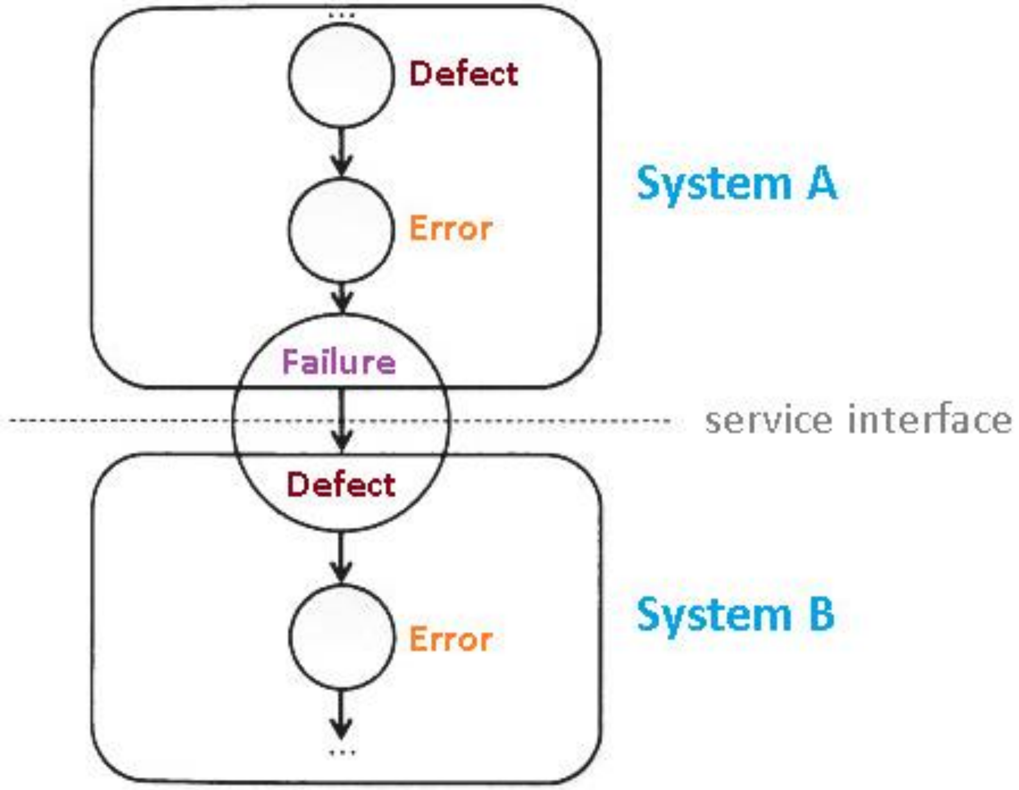
- Because
  - not all people could be good Test Analysts (and it would be waste to force strongly Convergent-thinking person on Divergent TA, or vice versa)
  - Multi-role QA‘s switch between Divergent (analysis) and Convergent (implementation) modes!
- Time consideration:
  - Switching to Convergent thinking: near-instant; natural=easy; quick immersion
  - Switching to Divergent thinking: 10-30 minutes; hard; slow „take-off“
- **→If possible, process-wise separate Divergent and Convergent QA activities**
- TA as subset of Automated testing: degradation of TA performance
  - Convergent „ to achieve“ thinking (Repo/Console control, high-level coding) mixed with
  - Divergent „why“, „what-if“ thinking (test analysis to identify individual Test Cases)
  - **→ time savings on automation phase, time (and quality) penalty on analysis phase**

Errors propagation

Don‘t mistake the effect for the cause

Defect → Error → Failure   cascades

- „Bug“ is wide, umbrella term
- Error you *observe* means that problem has reached an external interface – such visible, apparent error is called **failure**
- But actual root cause could be elsewhere: root cause is **defect**
- Between bug and it‘s manifestation could be many steps through which **error** spreads



- Why to distinguish? Because otherwise you could fall for the trap of:
  - searching for the defect at the module/step where the failure manifested... While it‘s really somewhere else
  - having invalid assumptions

Inspiration: finding the Pentium FDIV bug

In October 1994, Dr. Thomas Nicely noticed an errorenous calculation in his distributed computing system – a **failure**. One of the recreations is this:

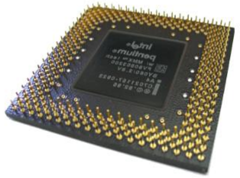
4195835.0 - 3145727.0*(4195835.0/3145727.0) = **0**	(Correct calculation result)
4195835.0 - 3145727.0*(4195835.0/3145727.0) = **256**	(Incorrect result = bug)

He went bug-hunting to isolate the root cause.

1. He first reviewed and refactored his code (3000 lines).
   But the bug wasn‘t there!

2. He discovered that the compiler optimizations enabled similar bug.
   But after disabling this „cause“, the bug returned again – this was an **error**!

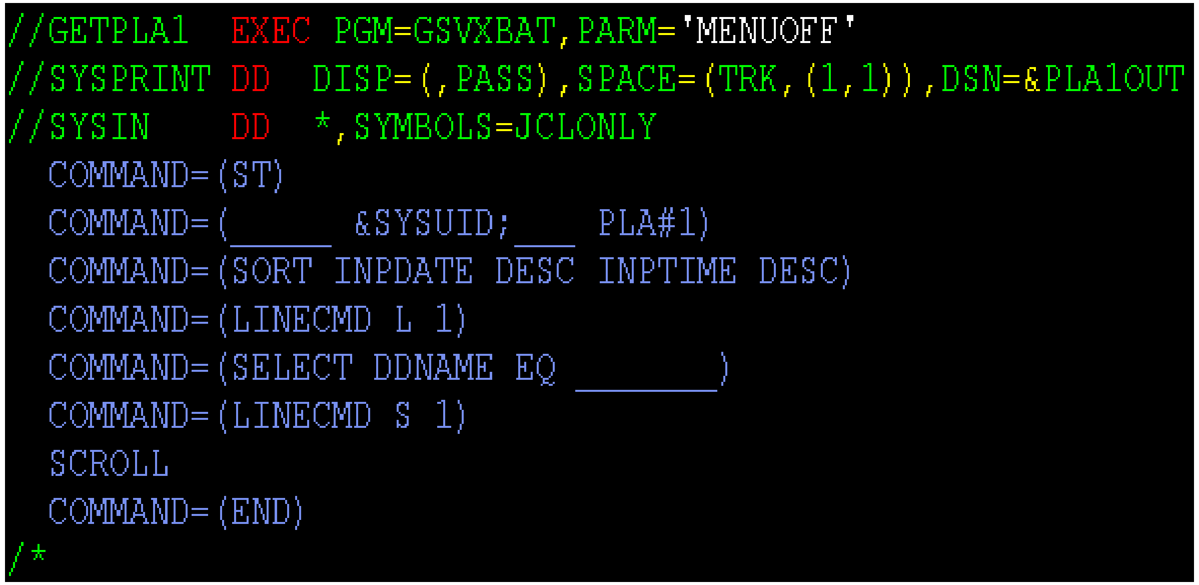
3. He reviewed the motheboard because some Neptune chipsets corrupted PCI data.
   But the bug appeared even on reliable, cross-checked chipsets!



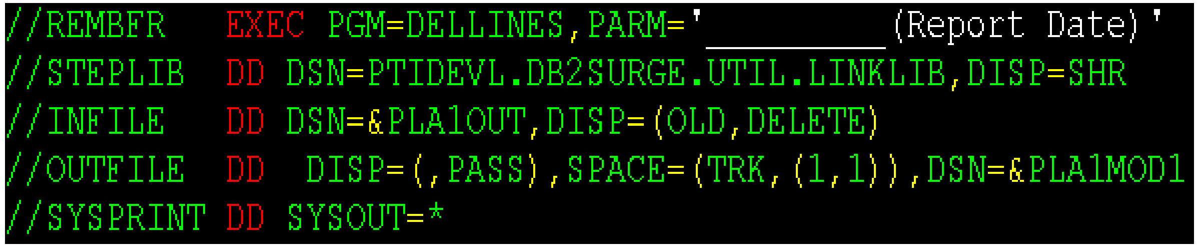
4. Then he disabled the FPU coproccessor in real-mode DOS.
   Disabled FPU → no failure. Enabled FPU → failure. This was the **defect**!

Lesson learned: what you think is a bug might be only another manifestation of the bug (=error), not the real cause. Fixing this „faux cause“ will just hide the real bug!

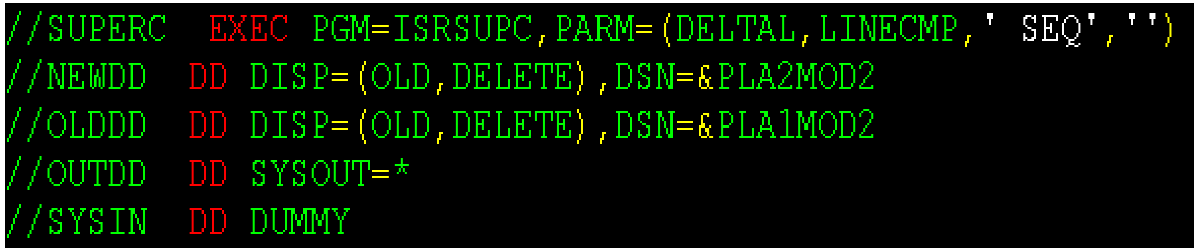
Defect → Error → Failure: data flow example of cascading error



```
//GETPLA1  EXEC PGM=GSVXBAT,PARM='MENUOFF'
//SYSPRINT DD  DISP=(,PASS),SPACE=(TRK,(1,1)),DSN=&PLA1OUT
//SYSIN    DD  *,SYMBOLS=JCLONLY
  COMMAND=(ST)
  COMMAND=(_____ &SYSUID;___ PLA#1)
  COMMAND=(SORT INPDATE DESC INPTIME DESC)
  COMMAND=(LINECMD L 1)
  COMMAND=(SELECT DDNAME EQ _______)
  COMMAND=(LINECMD S 1)
  SCROLL
  COMMAND=(END)
/*
```



```
//REMBFR   EXEC PGM=DELLINES,PARM='_________(Report Date)'
//STEPLIB  DD DSN=PTIDEVL.DB2SURGE.UTIL.LINKLIB,DISP=SHR
//INFILE   DD DSN=&PLA1OUT,DISP=(OLD,DELETE)
//OUTFILE  DD  DISP=(,PASS),SPACE=(TRK,(1,1)),DSN=&PLA1MOD1
//SYSPRINT DD SYSOUT=*
```



```
//SUPERC  EXEC PGM=ISRSUPC,PARM=(DELTAL,LINECMP,' SEQ','')
//NEWDD  DD DISP=(OLD,DELETE),DSN=&PLA2MOD2
//OLDDD  DD DISP=(OLD,DELETE),DSN=&PLA1MOD2
//OUTDD  DD SYSOUT=*
//SYSIN  DD DUMMY
```

Trainees were to fill in keywords in JCL steps:

- 1st step was to extract job output of a report to a temporary dataset.
- 2nd step was to delete all lines of the dataset from 1st step before the string „Report Date“.
- 3rd step was to compare output of the 2nd step with similar output.

Some people were getting JCL error at the 3rd step: a „dataset not found“ failure and RC=4 from the 2nd step.

However, the true defect was in the 1st step!

Root Cause Analysis

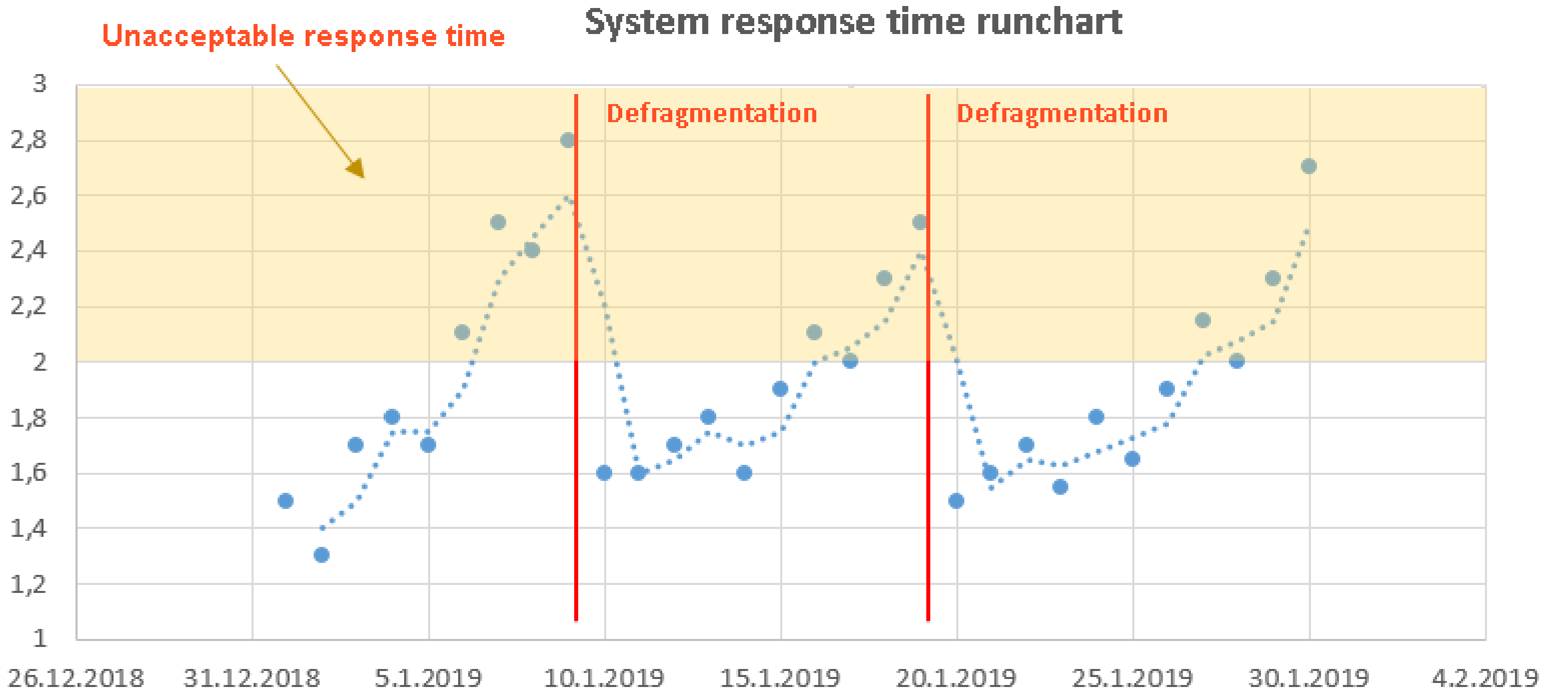
Find&fix the true cause, not a scapegoat

What is Root Cause and why target it?

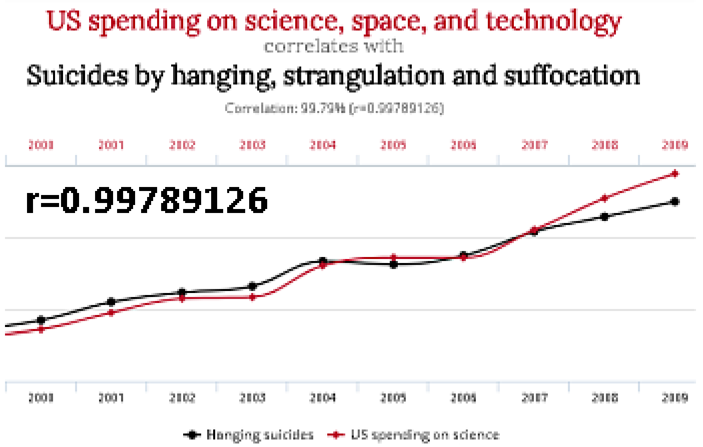
- Why?
  - Curing symptom instead of Cause only hides the problem (see Thomas Nicely Pentium FPU)
  - There can be multiple concurrent causes, removing just single may look like a fix but is not
  - More reliable method to address Root Cause can be found (eg. design change which renders it impossible for component to fail; avoiding the need for fauiling component; etc.)
- How?
  - RCA: systematic methods to decompose chain of events all the way to the original cause
  - „Storytelling“ how did the failure happen: how it all started and why it went wrong
  - Finding „just contributing factors“ instead of superficial („dumb workers“ vs. „insufficient instructions“)
  - **Goal and mentality: not punish, but prevent reoccurence of the failure**
- Methods
  - Fishbone diagram, Tree diagram
  - „5 Why‘s“
  - Run chart

Run chart

- Plotting of values in time, searching for correlation before failure
- Effective for finding „time causality RCs“, eg. insuffucient frequency of PM



- Basis of all „Machine Learning AI preventive analysis“
- Correlation ≠ implication! What if discounts are each 10d?
- See: http://www.tylervigen.com/spurious-correlations



5 „why‘s”

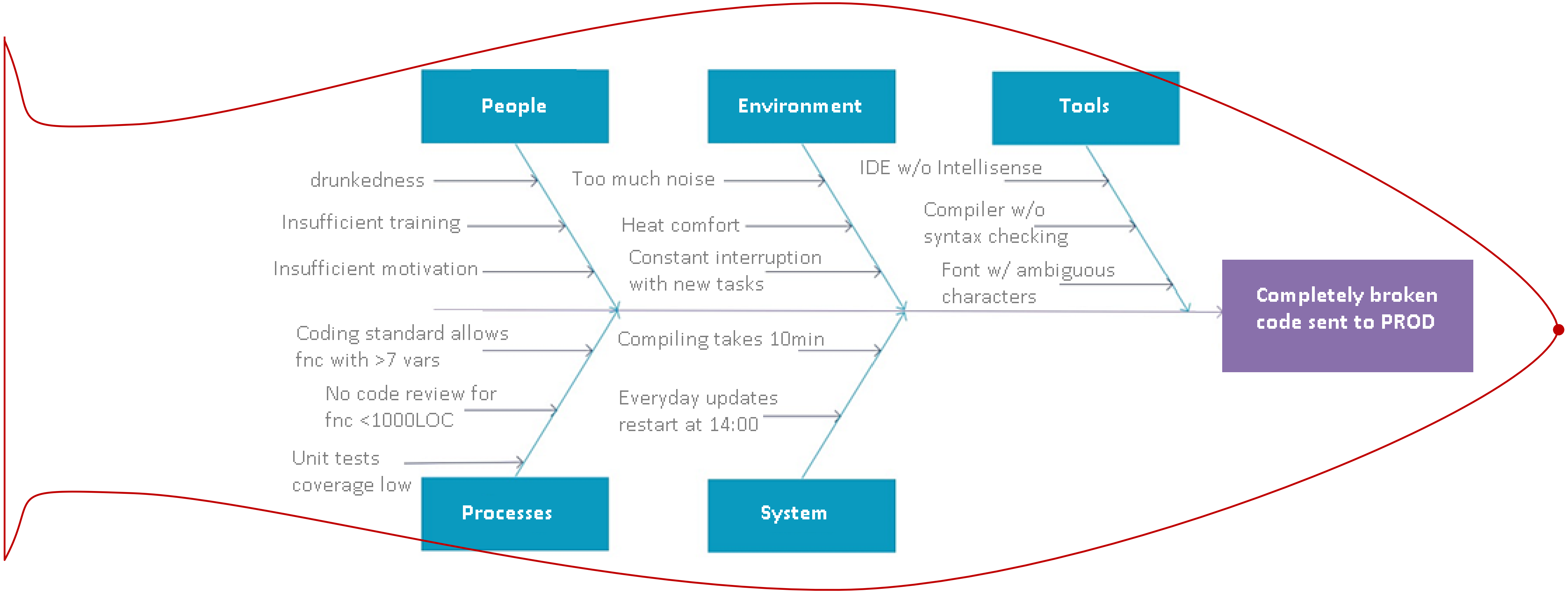
- Assumption: asking „why“ 5× usually gets you to the point

EFFECT		CAUSE
Worker injury	*because*	Fell
Fell	*because*	Slippers surface
Slippery surface	*because*	Oil leak
Oil leak	*because*	Seal ruptured
Seal ruptured	*because*	Replacement interval not met

- Pro‘s
  - Often works... IF the root cause is 1) single, 2) simple
  - Not too superficial, nor too exhaustive... Average.
- Con‘s
  - Cannot work / misleading when root cause happens by coincidence of more factors

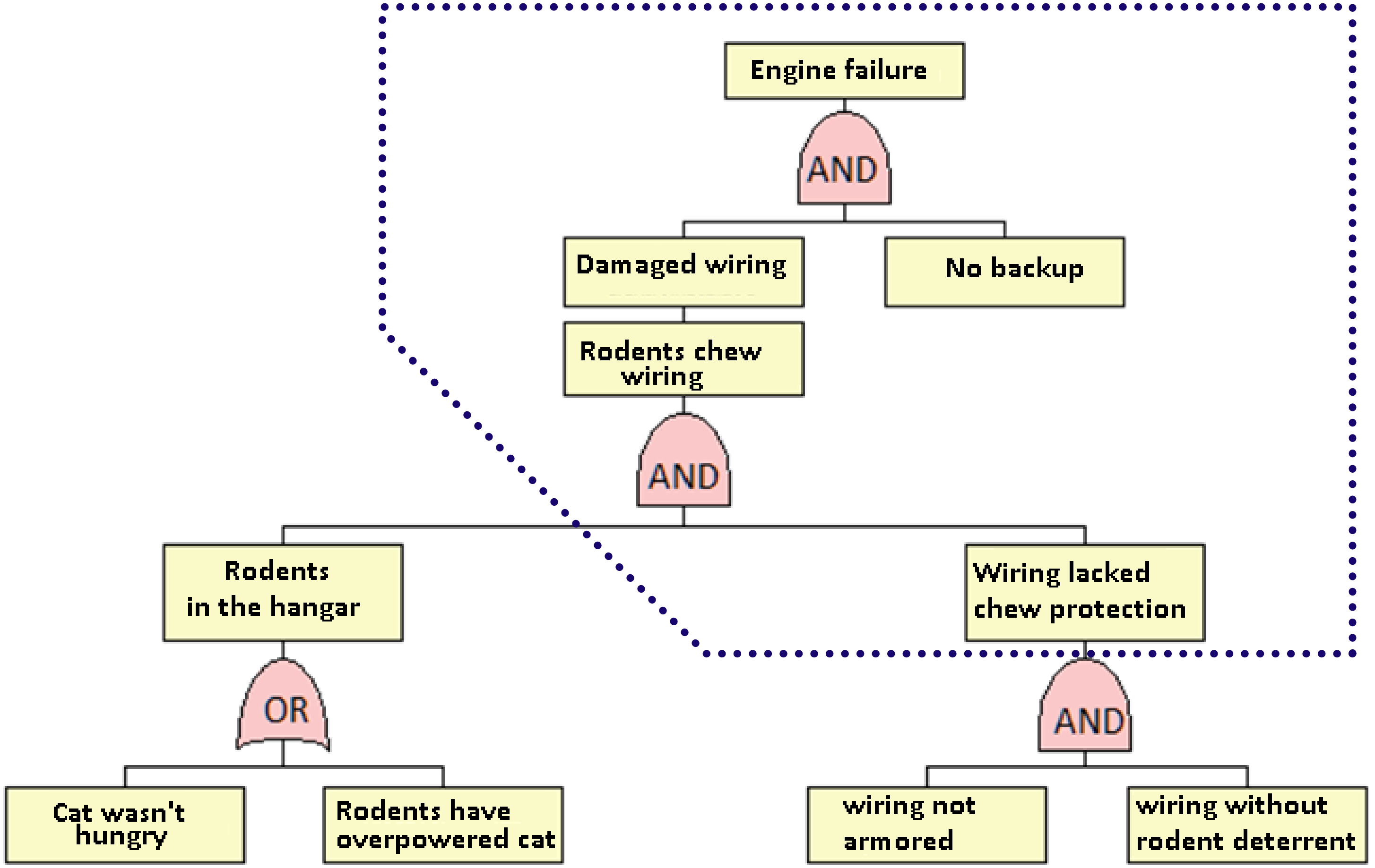
Fishbone diagram (Ishikawa)

- Focus on examining different catherogies of causes (people/design/tools/processes/...)
- Every cathegory = cheat sheet for examining hypothetical causes to be examined
- All reasonable causes are examined, real causes identified, systematically addressed


People
Environment
Tools
drunkedness
Insufficient training
Insufficient motivation
Too much noise
Heat comfort
Constant interruption with new tasks
IDE w/o Intellisense
Compiler w/o syntax checking
Font w/ ambiguous characters
Completely broken code sent to PROD
Coding standard allows fnc with >7 vars
No code review for fnc <1000LOC
Unit tests coverage low
Compiling takes 10min
Everyday updates restart at 14:00
Processes
System

Tree diagram

- Combining the best from „5 why‘s“ and Fishbone: any number of levels, categories
- Plus combination of causes, plus logical operators: AND OR



Assumptions and ambiguity

The biggest obstacles to Validation

Always challange assumptions. Especially „cannot happen“/„will always“

- Convergent thinking effect: developers *must* use assumptions to code
- Sometimes, these assumptions are implicit yet unsubstantiated
  - „The system will never lose electrical power.“
  - „No user would ever use characters like «{» or «]» in his user name.“
- Often, these assumptions are fragile or unhealthily dependent
  - „Negative value on input could never occur, function XYZ prevents that.“
    - What if function XYZ is defective? What if the input circumvents function XYZ?
  - „Input will have value of either «1», or «2».“
    - What if «3» is added in the future? What if «-1» is passed due to computation error?
  - „The system will never run for longer than *n* days“
    - Remember the Patriot PAC-3 in Kuwait

Always challenge assumptions. Also implementation assumptions.

- ISTQB standard-issue example: developing an e-mail system

  „*As a user, I want to send and recieve e-mails so that I could communicate with my family and co-workers*“

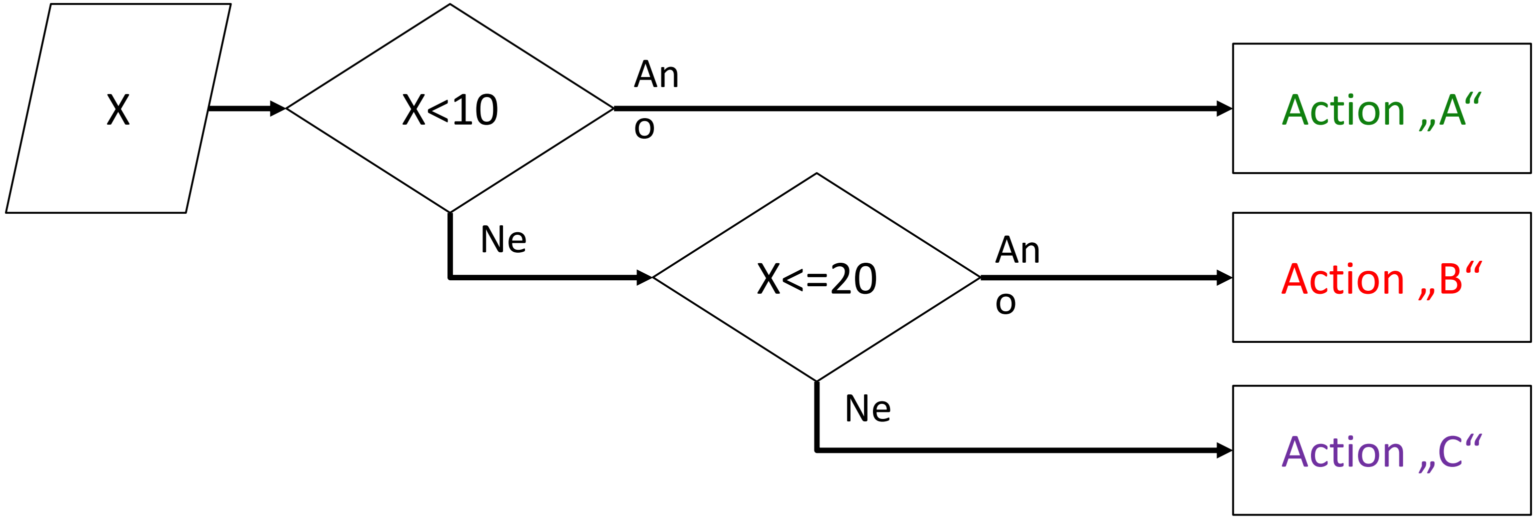
- Simple use-case, many missing answers
  - Encodings support?
    - 7-bit ASCII + Base64?
    - Unicode?
    - Native Unicode or encoding schemes & transfer-encoding, HTML entities?
  - Attachments?
    - Types?
    - Size?
  - Content format?
    - Plain-text?
    - HTML?
    - RTF?
  - (...)

Handling the inputs

Equivalence partitioning, Dependency Islands, All-pairs testing Ortogonal arrays

Equivalence Partitions and Boundary Values Analysis

- Imagine your software has flow with several important code paths dependent on integer variable X, and input could be anything within -100 and +100:



X

X<10

Ano

Action „A“

Ne

X<=20

Ano

Action „B“

Ne

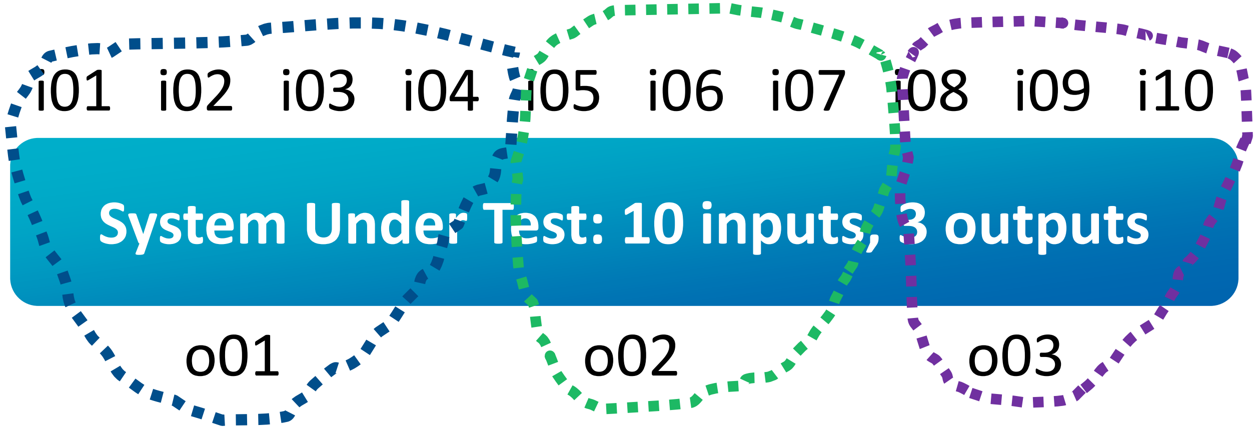
Action „C“

X value	Action
-5	A
9	A
10	B
11	B
19	B
20	B
21	C
100	C

- All values within „switch condition“ belong to identical **EP** = like 1
- Each „switch condition“ (x=10/20) represents a „boundary value“ (**BV**) which you test
- But you also need to test above and below BV to cover errors in „<=„ and „>=„ logic
- You also need to test negative values and variable boundaries (int: 32768±1)

Dependency Islands within Equivalence Partitions

- If you have more input, you have to multiply inputs with their EPs:
  (№ of Input 1's EPs) × (№ of Input 2's EPs) ... × (№ of Input N's EPs)
- 10 inputs with 4EPs each = 4 × 4 × 4... × 4 = 4^{10} = 1,048,576 tests
- Only combinations of inputs affecting shared output = **Dependency Islands**



- (i01 × i02 × i03 × i04) + (i05 × i06 × i07) + (i08 × i09 × i10) = $4^4 + 4^3 + 4^3$ = 576 tests
- Difference: 10^{23} (ranges without EPs) vs. 10^6 (just EPs) vs. 10^2 (DIs)
- Warning: gray-box testing – could trigger incorrect assumptions!

Ortogonal arrays: all-pairs testing for dependent variables

- What if you cannot use Dependency Islands, but have few inputs with few EPs?
- Brute force each-with-each has many redundancies (i01 × i02 as well as i02 × i01...)
- Studies: most defects caused by single-mode or double-mode faults (1|2 vars only)
- → Not „each with each“, but „each with exactly one else“
  1. pair all the inputs: Binomial coefficient $\binom{n}{k} = \frac{n!}{k!(n-k)!}$
     where n = pairs, k = 2 (eg. 3 inputs = 3 pairs, 4 inputs = 6 pairs)
  2. Ortogonal array with columns = inputs, rows = EPs per each input
  3. Fill sorted pairs of inputs to avoid missing any combination pair, using ortogonal arrays
- Ortogonal arrays hard to construct
  - Template: support.sas.com/techsup/technote/ts723_Designs.txt

i01.EP1	**i02.EP01**	**i03.EP03**
i01.EP1	i02.EP02	i03.EP02
i01.EP1	i02.EP03	i03.EP01
i01.EP2	i02.EP01	i03.EP02
i01.EP2	i02.EP02	i03.EP01
i01.EP2	i02.EP03	i03.EP03
i01.EP3	i02.EP01	i03.EP01
i01.EP3	i02.EP02	i03.EP03
i01.EP3	i02.EP03	i03.EP02

Fail-safe, Fault-tolerant, Single Point of Failure

It‘s important how systems fail

Single Point of Failure

- Failure of a component directly causes failure of the entire system
- Architecture variants:
  1. failure of a component independently causes failure of system
     (eg. overflow→overwrite of RAM, breakdown of fuel pump→fuel tank breach and explosion)
  2. failure of a component propagates an error which causes failure of subsequent components and system
     (eg. corruption of user data, which cannot be recovered)
- Manifests when:
  - there is no backup/check of an important component
    (Boeing 737MAX, LA610, 29.10.2018: MCAS drawing on data from single AoA sensor)
    OR
  - failure of an important component is not / cannot be detected
- Discovery in QA
  - Fault Tree Analysis – any Minimal Cut Set, which doesn‘t contain „OR“, represents a SPoF

Recovery from intermittent error

- Component could be designed to recover from intermittent, random errors
- Architecture varians:
  1. simple „retry“ – ignore error, wait, retry
     (eg. simple loop to repeat instruction/function again)
  2. „error masking“ – error is thrown away and not propagated „downstream“
     (eg. threshold which cuts-off spikes from analog input data; router which deletes bad packets)
  3. rollback-to-savepoint – revert to earlier snapshot of correct data
     (eg. „recovery points“ in Windows OS, ImageCopy of DB)
- Requirements:
  - system is resistant against partial data loss
    (from savepoint to rollback; thrown away bad data; ignored data)
  - error is temporary and random, not systematic or persistent
  - error doesn‘t manifest too often
    (it‘s not possible to ignore/drop 50% of packets on network and still provide connection/service)
- Architecture risks:
  - endless loop waiting for recovery – a watchdog circuit/function is required

Fail-safe

- Failure of a component is isolated, contained and does ***not*** lead to system failure
- Architecture variants
  1. Failed component shuts down contained and transfers function to existing backup
  2. Failed component restarts into default („semi-fail-safe“, but still semi-SPoF)
- Functional requirements
  – Detection of error or failure
  – Complete isolation of failed component from system
    (eg. fuel pump explodes, but firewall prevents schrapnels from rupturing or igniting the fuel tank)
  – System is resilient to outages before backup is online, ***OR*** system uses „hot backup“
- Architecture risks
  – Fault to detect Failure
  – Fault to transfer function to Backup (backup N/A, backup not „woken up“, cannot transfer function)
  – Fault to provide resources for contained shutdown of failed component
    (parking of HDD heads during power outage, Fukushima reactor fuel cool-down)
  – Common Component Faults

Fault-tolerant

- Parralel processing in multiple identical components
- Outputs are compared, deviation from „democratic majority“+tolerance is considered faulty and the deviating component is flagged and shutdown
- Architecture variants:
  1. independent evaluation of multiple sensors
- Functional requirements:
  - each fault-tolerant component is completely independent „upstream“ (no shared sensor, data source)
  - odd number of fault-tolerant components to prevent draw → minimum of 3
- Architecture risks:
  - „democratic majority“ of components is wrong
    (Airbus 320, LH1829, 5.11.2014: frozen AoA_1=4,2°, frozen AoA_2=4,6°, functioning AoA_3 „lost the vote“)
  - the one commands-issuing component has time to propagate error before it‘s flagged as deviating
    (Airbus 330, QF72, 7.10.2008: spikes in AoA_1 shorter than 8sec threshold → not-suppressed)
- Discovery in QA
  - FTA needs to confirm complete upstream redundancy

Common Component Failure

- Backup is identical as Primary in design/parameters and fails for same reasons
- Manifests when:
  1. the components were unfit for the use (too weak, too low memory, insufficient material strength)
  2. all components were identically incorrectly installed (too small allocation of HW resources...)
  3. all components were identically exposed to wrong maintenance (bad patch, copypasted config file)
  4. all components were exposed to the identical rush environment (large data flow, high temperatures)
- Manifests when:
  - variant №4:
    - Any Fault-tolerant system
    - Fail-safe system with Hot backup
  - variants №2-3: system is „destined to fail“ due to external influence
  - variant №1: component shouldn‘t have been used in the system
- Architecture risks:
  - probabiility *n* of *m* components failing independently on each other = $P_{indep}=n^m$
  - probability n of m identical components failing commonly = just $P_{common}= n \times P_{CCF}$ = larger by magnitude
  - eg. if m=3 components, $n=10^{-3}$, $P_{CCF}=10^{-2}$ – then $P_{indep}=10^{-9}$ but $P_{common}=10^{-5}$ = 10,000× more probable

Integrity levels

Figure out how throuhgly you need to test

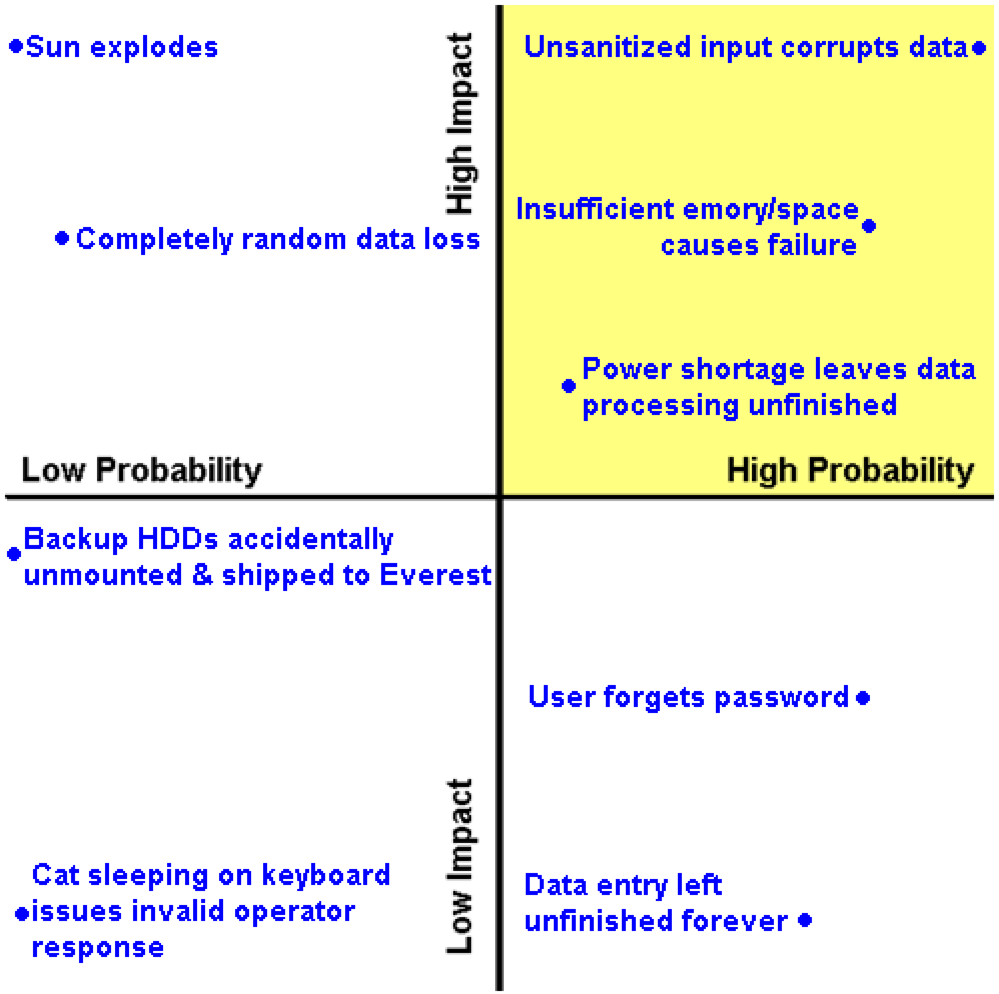
Consequence-based required integrity levels

If the software fails:	Integrity level
No mitigation is possible and catastrophic consequences will occur: • Loss of human life • Complete system or mission loss • Loss of system security and safety • Extensive financial, social or environmental loss	4
Partial-to-complete mitigation is possible but critical consequences will occur: • Major and permanent injury • Partial loss of mission or major system damage • Major financial, social or environmental loss	3
Complete mitigation is possible but marginal consequences. • Moderate injury or illness • Degradation of secondary mission • Moderate financial or social loss	2
Mitigation is not even requiered since only negligible consequences will occur: • Minor impact on system performance • Operator inconvinience	1

PROD data loss

Severity-Probability chart: what to focus on

- Divergent „*what-if*“ mindset and FMEA identify 100s possible problems
- „*Exhaustive testing is impossible*“: there are more code paths in Unix than atoms in our galaxy
- Solution: 2D probability-severity graph
- You must test for upper right quadrant.
- Then top half, then right half.
- But how to figure out which all quadrants must be covered?



High Impact

Low Impact

Low Probability

High Probability

•Sun explodes

•Completely random data loss

Unsanitized input corrupts data•

Insufficient emory/space causes failure•

•Power shortage leaves data processing unfinished

•Backup HDDs accidentally unmounted & shipped to Everest

•Cat sleeping on keyboard issues invalid operator response

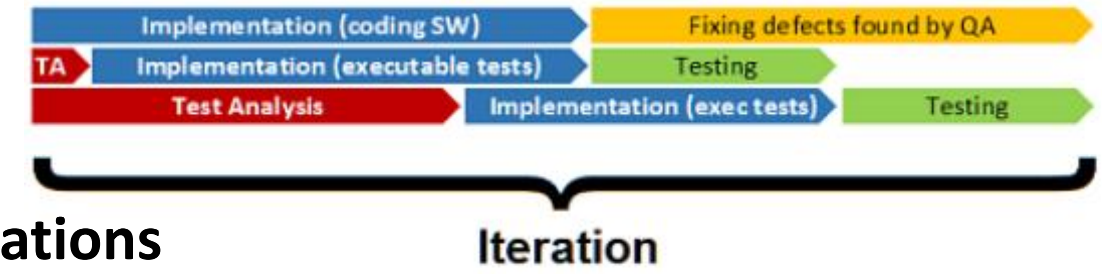
User forgets password•

Data entry left unfinished forever•

QA versus Agile, TDD, „safe-to-fail“

Few words on Level 3 / 4 QA and TA in Agile methodologies

- **Processes are here *for* you (not you for the processes)**
  - Processes cannot compensate for unskilled or complacent humans („ISO 9000 myth“)
  - Processes are not goal on their own („Death by processes“)
  - However, highly skilled people without processes can fail: forget or skip what needs to be done
  - Agile Manifesto: „individuals over processes“, not „no processes“ → **hack** processes to your needs!
- **Finding enough time for analysis is challenge, but *could* be overcome**
  - Need to prioritize, use Risk-based testing (Integrity Levels, Fn list → FMEA → Critical Fn list)
  - „Individuals over processes and tools“ → modify tools – lightweight/partial FMEA, FTA, STT...
  - If you have >1 QAs, you can paralelize tests and TA:
  - Design & Logistic sprints („people over processes“)
  - Expert TA shared between teams akin Architect


Implementation (coding SW)
Fixing defects found by QA
TA
Implementation (executable tests)
Testing
Test Analysis
Implementation (exec tests)
Testing
Iteration

- **Don‘t forget the *mindset & skillset* considerations**
  - Nor Devs nor „Basic Testers“ have skills for „Level 4“ QA and Test Analysis – you need proffesionals
  - frequent/cyclic switching between Divergent/Convergent thinking inefficient, detrimental to coverage
  - Both are „processes“ considerations – how, when and with whom you get the work done

TDD: Unit Tests are good start, but never enough

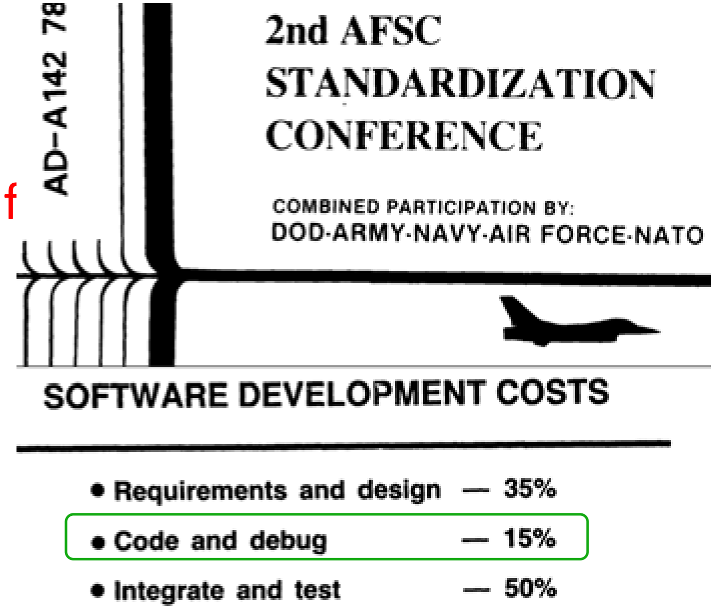
- TDD: „*first write tests, then develop SW to pass these tests*“
- if it takes >week to do TA and then some to write TCs, will DEVs sit idly? No...

➡ IRL, „TDD“ means „developer writes Unit tests and then code to pass that UTs“

- Unit Tests positives:
  - lead to „Defensive programming“
  - greatly increase code quality (psychology)
- Unit Tests limitations:
  - cannot be exhaustive: you mostly cannot say „∀ input x, f(x) works correctly“ (unassumed vars)
  - test smallest artefact → you can never say „if all *artefacts* pass UT, *system* works correctly“
  - inherently can‘t Validate design, can‘t verify DEV‘s assumptions, can‘t test INTEG or ENV
- Unit Tests are inherently „Level 2: positive testing“ or „Level 3-“ QC ➡ insufficient
- Unit Tests cannot be considered replacement even for QC, never for QA

Killing the false myths: QA vs. Culture of improvization and skill-lessness

- *„robust approach to testing, which is costly (...) TDD ... your codebase includes robust unit-test coverage. This guarantees that the system will work as the developer intended"* –Devskiller.com
  „We cannot say the function is correct for *any* input. We cannot say that if all modules works, the system works together."
- „Move fast and break things" –Mark Zuckerberg
  „Move fast with stable infra" –the same Mark Zuckerberg later on
- „In Agile world, developers became testers ... developers perform tests and testers perform development" –Ray Arell, Agile Alliance
  „if ¬(∀Dev→Dev ≈ Architect), then ¬(∀Dev ≈ Test analyst)"
- „Our SW will never be mission-critical, doesn't need big QA"
  VS. Chrysler infotainment, CIA comms web-app, Metromile hackable dongle
- „Agile says to quickly move forward via trial-and-error"
  „Take small steps ... adjust based on what you learned ... take the path that makes future change easier" –Dave Thomas, Agile MF co-author



AD–A142 78

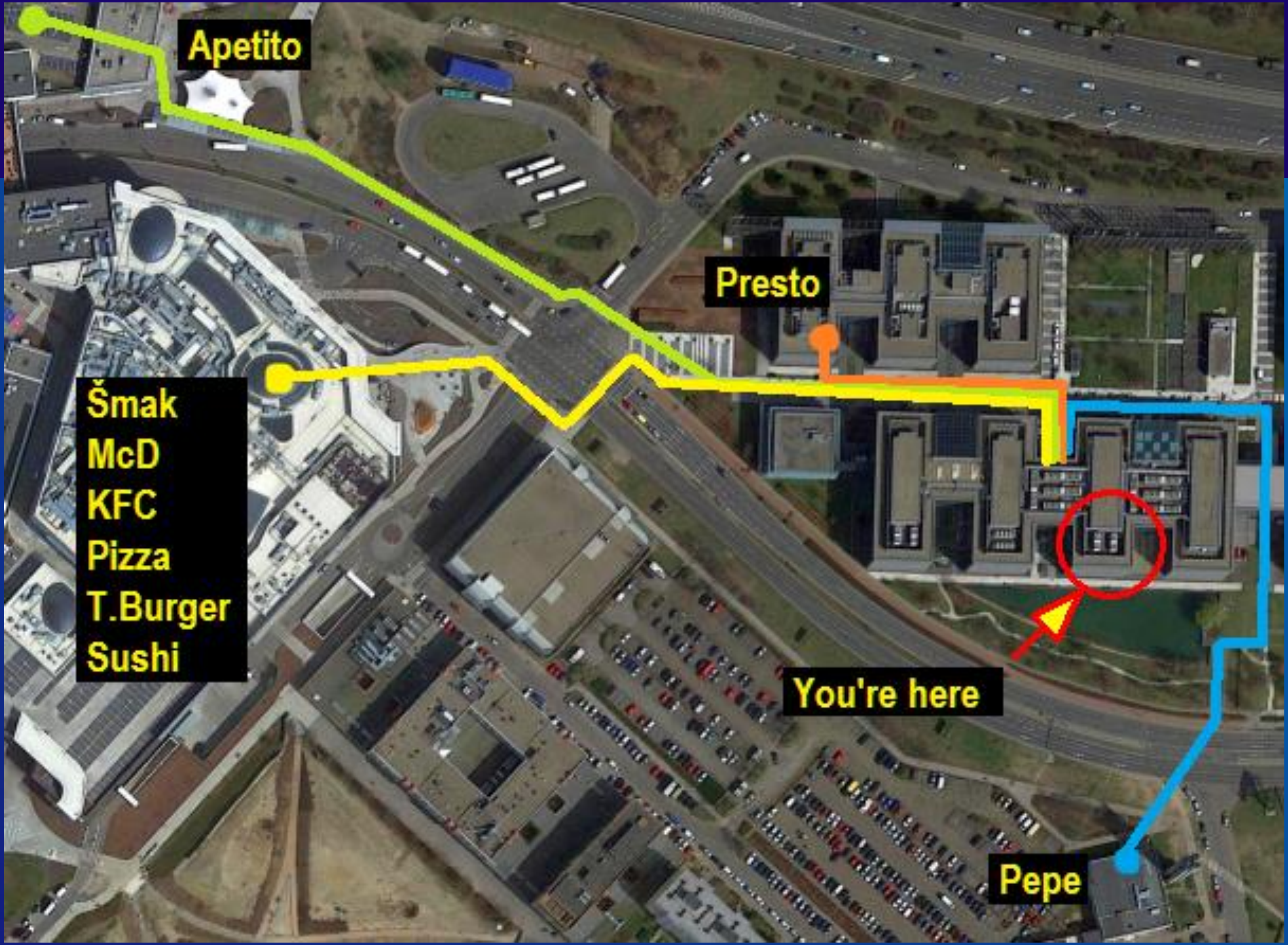
2nd AFSC STANDARDIZATION CONFERENCE

COMBINED PARTICIPATION BY:
DOD·ARMY·NAVY·AIR FORCE·NATO

SOFTWARE DEVELOPMENT COSTS

- Requirements and design — 35%
- Code and debug — 15%
- Integrate and test — 50%

Lunch break

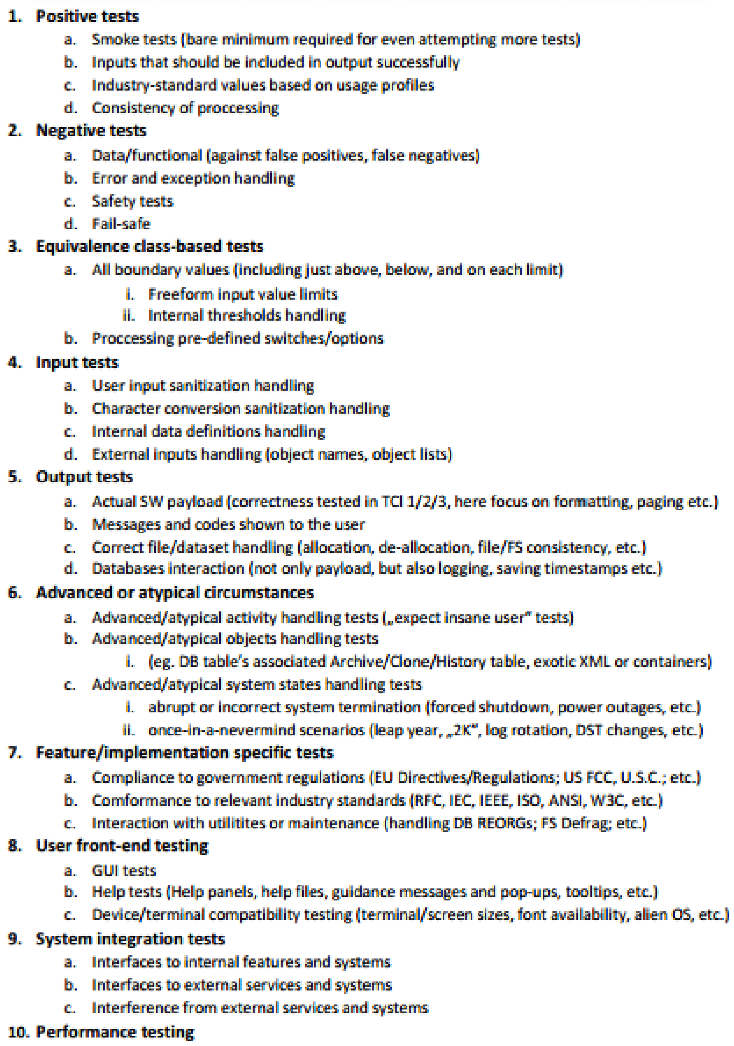

Apetito
Presto
Šmak
McD
KFC
Pizza
T.Burger
Sushi
You're here
Pepe

Test Classes

QA engineers best friend, checklist and cheat-sheet

Test Classes: cheat sheet to remember&organize your TCs

- QA includes lot of test topics/techniques for each Maturity level
- Even we haven‘t went through all of these
- So how to:
  - Not forget to consider any important test topic?
  - Avoid duplicate testing?
  - Not get lost in all the Test Cases and organize it?
- Answer: with **Test Classess**!
- Officially (per IEEE 829-2008):
  - „A designated grouping of test cases“
  - „Summarize the unique nature of particular level of test“
- Practically could be expanded into:
  - Checklist and Cheat-sheet for Test Analysis
  - Organizing structure („spine“) for the HLTCs identified



1. **Positive tests**
   a. Smoke tests (bare minimum required for even attempting more tests)
   b. Inputs that should be included in output successfully
   c. Industry-standard values based on usage profiles
   d. Consistency of proccessing
2. **Negative tests**
   a. Data/functional (against false positives, false negatives)
   b. Error and exception handling
   c. Safety tests
   d. Fail-safe
3. **Equivalence class-based tests**
   a. All boundary values (including just above, below, and on each limit)
      i. Freeform input value limits
      ii. Internal thresholds handling
   b. Proccessing pre-defined switches/options
4. **Input tests**
   a. User input sanitization handling
   b. Character conversion sanitization handling
   c. Internal data definitions handling
   d. External inputs handling (object names, object lists)
5. **Output tests**
   a. Actual SW payload (correctness tested in TCl 1/2/3, here focus on formatting, paging etc.)
   b. Messages and codes shown to the user
   c. Correct file/dataset handling (allocation, de-allocation, file/FS consistency, etc.)
   d. Databases interaction (not only payload, but also logging, saving timestamps etc.)
6. **Advanced or atypical circumstances**
   a. Advanced/atypical activity handling tests („expect insane user“ tests)
   b. Advanced/atypical objects handling tests
      i. (eg. DB table’s associated Archive/Clone/History table, exotic XML or containers)
   c. Advanced/atypical system states handling tests
      i. abrupt or incorrect system termination (forced shutdown, power outages, etc.)
      ii. once-in-a-nevermind scenarios (leap year, „2K“, log rotation, DST changes, etc.)
7. **Feature/implementation specific tests**
   a. Compliance to government regulations (EU Directives/Regulations; US FCC, U.S.C.; etc.)
   b. Comformance to relevant industry standards (RFC, IEC, IEEE, ISO, ANSI, W3C, etc.)
   c. Interaction with utilitites or maintenance (handling DB REORGs; FS Defrag; etc.)
8. **User front-end testing**
   a. GUI tests
   b. Help tests (Help panels, help files, guidance messages and pop-ups, tooltips, etc.)
   c. Device/terminal compatibility testing (terminal/screen sizes, font availability, alien OS, etc.)
9. **System integration tests**
   a. Interfaces to internal features and systems
   b. Interfaces to external services and systems
   c. Interference from external services and systems
10. **Performance testing**

Test Classes types: Positive tests

a) **Smoke tests**
Subset of tests verifying only the most basic functionality (like level 1 verification testing); why? Because if even the most basic tests fail, there is no point in more testing – the SUT needs extensive fixing and re-test

b) **Inputs that should be included in output successfully**
Testing of valid and positive inputs: eg. in DB, when you SELECT LIKE(A*), SUT returns records „Amy“, „Amanda“ and „Andrej“. Complex test coverage against false-negatives, missing or incorrect calculations/data.

c) **Industry standard values based on usage profiles**
You have standardized data exchange formats (UNIFI/XML, Exactis/JSON, etc.), syntax formats (W3C, IEEE, etc.), containers (Docker). Your product should handle even the most extensive variants of standards since they are allowed.

d) **Consistency processing**
Run SUT several times with relevant inputs unchanged. It shall not change despite changes in „irrelevant inputs“ (system time, workload, etc.)

Test Classes types: Negative tests

a) **Negative outputs**

Outputs which should NOT be included. This isn‘t destructive testing, this is positive testing with double negation: when the SUT should NOT do something, we prove it really DOES NOT. Eg. DB with filter on „A*“ shall NOT return „Dave“.

b) **Failure mode tests**

Verification that when product fails, it behaves correctly. Not destructive.

I. **Failure Mode sanitization**: when component B depends on component A, what happens when component A fails? Is it sanitized in component B?

II. **Error messages**: you need to test that when something fails in SUT,
a) error message is issued,
b) appropriate/correct error message is issued,
c) no false-positive error msgs.
Example: IBM Db2 -405: numeric constant out of range for 9x10E18 while range is 7x10E75

c) **Fail-safe**

If the SUT contains Fail-safe components, tests must verify all 3 principles of Fail-safe: fault detection, fault isolation, and backup

Test Classes types: Advanced & Destructive tests I

a) Advanced/atypical activity

I. **„Stateful“ scenario testing**: use a sequence of steps to bring the SUT into an atypical but allowed state, and only then start testing other inputs (=SUT starts in a different state than typically)

II. **Using allowed but rare utilities, processing options/settings**: concern especially with interoperability, reporting, auditing. Compatibility modes, runtime „-parms“...

III. **Rollbacks, incident recovery**: rare but extremely important; very hight potential for data corruption caused by state-transitions going forth-back-forth...

IV. **Concurrent activity**: something else runs in parralel with SUT sharing the same data, connection etc., there could be conflicts (subjectivelly intermittent errors)

V. **State transition testing**: real STT using SUT ST models, especially back-and-forth transitions and „transition leaps“, and race conditions or state-transition-interruption

Test Classes types: Advanced & Destructive tests II

b) Advanced/atypical objects handling

I. **Using allowed but rare object parameters, variants, options**: similar to previous test sub-class but focus on objects, not procedures. Eg. file attributes, physical parameters (line length, paging/segmentation settings etc.); OPP object instances of overridden types; anything non-default and exotic.

II. **Advanced object types**: eg. DBs special table types (eg. LOBs, Archive and History tables, temporary virtual tables instantiated from selects...); exotic container types or variants; ancient or exotic but allowed formats (.CAB archive on Windows? HTML 4.x deprecated features in 2020? Japanese/Asian formats?)

Test Classes types: Advanced & Destructive tests III

c) Advanced/atypical (host) system states handling

I. **Once-in-a-nevermind events:** DST changes. Leap year. Log rotation. „Y2K“. File system migration. Host system upgrade. Host system downgrade. Any irregular event which may affect the SUT but happens so rarely that it could be severely undertested – until the „wrong“ combination of input parameters occurs.

II. **Host system in degraded mode/recovery mode/limited capacity**: eg. banking core servers on backup power with reduced cooling=workload capacity; production database in recovery mode trying to recover lost data; etc. Emergencies where correct functionality is critical but resources are limited both performance-wise and SW-wise.

Test Classes types: Advanced & Destructive tests IV

c) **Fault insertion tests**
Testing how does the system detect and handle when the data it uses become corrupted. Eg. damaged packet received; or file contents garbaged by HDD faulty sector; or half-written config file is attempted to be read.
Example: infinite loop on Db2 IC damaged via utilities

d) **Other relevant failure modes**
A placeholder for other advanced destructive tests covering potential threats detected by FTA, FMEA, or RCA of customer defects, which however didn‘t fit anywhere else (or it would be pointless bureocracy to split it)

Test Classes types: Equivalence-partitions-based tests

a) **Boundary values testing**

  I. **Freeform input values limits**
  Testing how the length and size of variables entered by user is handled (eg. that you cannot enter more digits/characters than allowed, that you cannot enter value higher than allowed (eg. 32769 for integer)

  II. **Internal EP/BVA testing**
  Path testing within the code depending on user inputs (EPs+BVAs depending on the switches within the code regarding freeform values)

b) **Processing pre-defined switches/options**
Again EP/BVA testing, only this time limited just to pre-defined paths (eg. you have checkboxes, drop-down menus and other enums)

c) **Stress testing**
When you say the SUT could handle files with 10^6 rows, or run for 100 days consecutively, you go and actually try that if you stress the SUT so, it doesn‘t fail

Test Classes types: Input tests

Not „what you do“ with inputs (EP/BVA) but „how do you handle“ them

a) **Human Operator Error Tests**

I. **Input sanitization handling**
How are unexpected/undesired inputs handled. Eg. characters instead of digits; apostrophes; Unicode characters from range>255; brackets; periods, but also and crucially „TAB“ (\t), newline (\r\n), and null (\n or \0) characters absolutely need tests

II. **Character conversion handling**
Most systems could be accessed with multiple encodings; how do you handle conversions, especially of critical characters? Eg. in IBM DB2 for z/OS natively operates with CodePage 500, everyone in EMEA uses it in CP037, and then „<![CDATA[„ becomes „<]¬CDATA¬“;
how about „odd“ encodings like 7b-ASCII, UTF-7, or Base64?

b) **Internal data definitions handling**
When you use constants or pseudo-constats which are matched, is the same really being matched everywhere the same? Especially important during regression testing where the constants may change...

c) **External inputs handling**
Loading inputs from outside SUT itself. Config files (even your own – test different/old versions, user-made-edits, etc); Object lists passed by interoperable systems; etc.

Test Classes types: Output tests

a) **Actual SUT payload**

Not „what do you make“ but „how does it look“. Line-wrapping, print paging, paragraph breaking, but also crucially completeness of displayed data

Example: C# RichTextBox and maximum of 2,147,483,647 characters vs. SQL generator

b) **Messages and codes shown to the user**

Like Error messages handling, but here it‘s about informational, advisory and warning messages.

c) **Correct file handling**

Allocating, de-allocating, file locks are being enabled and removed, file/FS consistency isn‘t being corrupted, etc.

d) **Database interactions**

If you‘re saving the outputs, logging or doing anything with DB, verify the SUT writes correct data and handles DB considerations incl. locks and timestamps of data OK and consistently

Test Classes types: Feature/implementation specific tests

a) **Compliance to mandatory regulations**
If you‘re developing SW for aerospace, your SW needs to comply with DO-178C and/or MIL-STD-1345B. If you‘re processing private informations, you must comply with GDPR. Etc. In either case, state/EU/Federal regulations require additional special test cases.

b) **Compliance to relevant industry standards**
If you claim that the SUT „supports“ Unicode, XML, WiFi..., these are standards (RFC, IEC, IEEE, ISO, ANSI, W3C...) with many small features and catches and you need to test whether you fully support all these, otherwise you lie and can be sued.

c) **Interaction with maintenance or utilities**
If your SUT could be influenced by utilities such as Defragment, DB Reorg etc., or maintenance activities, you need to test surviving them.

Test Classes types: User front-end tests

a) **GUI tests**
Testing effort of it‘s own focused on front-end; it doesn‘t need so much Test Analysis, but may benefit from Ortogonal arrays all-pairs.

b) **Help tests**
Often forgotten or outdated, help documents, help panels and especially interactive help should reflect all current behavior and features.

c) **Device/Terminal compatibility**
Users may use wide variety of screen resolutions (eg. Lubuntu 800x723), screen orientations („portrait“/“landscape“), screen combinations (extended desktop, multi-monitor setup), devices (desktop, server with reduced multimedia support, handheld devices, text-only devices for blind people and UNIX consoles) etc. Do you really support them all?

Test Classes types: System integration and interoperability tests

a) **Interfaces to internal features and system**
Integration testing with things your SUT internally needs to function, or modules which are shared with other systems

b) **Interoperability with external services and systems**
If your SUT works stand-alone, but is designed to either process data generated by other system(s) „upstream“, or generate data processed by other systems „downstream“, double-check that all systems in the data chain are on the same page

c) **Interference from external services and systems**
When your SUT is stand-alone, but isn‘t the only system on a host, could other systems influence it, interfere with it? Eg. locking needed resources, overflowing into your SUT‘s memory, etc.?

Test Classes types: Performance testing

a) **Comparison testing**
   If you compare the newly developed SUT version with older, didn‘t performance significantly degrade? Are the users really willing to sacrifice performance for new features? What about competition‘s performance?

b) **Load testing**
   Isn‘t system operation near it‘s maximum workload unbearably slow?

c) **Consistency testing**
   Doesn‘t SUT noticebly slow down witch each chunk of data added? (Sometimes it‘s good to „normalize“ via artificial timers so that customers experience constant response rates instead of gradual slow-down)

d) *(~~**Stress testing**~~ might look as kind of performance testing but belongs to boundary values testing because you‘re testing ability to handle, not the performance in time)*

Demo: how Test Classes help QAEs

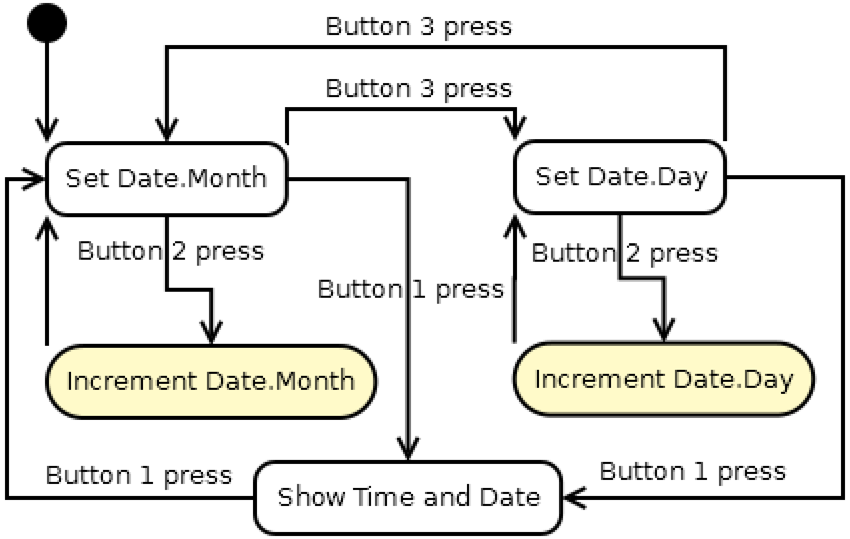
- Standard QA training excercise „How would you test an electric keetle“
- Typical answers, coverage of Test Cases limited to few:
  - **Temperature**: The water should be boiling to ensure maximum tea flavor. Do you assume that just because the kettle switches off that this means the water is at 100C?
  - **Safety**: If there is no water in the kettle, does it still heat up?
  - **Performance**: How quickly does it boil?
  - **Load**: Is it going to be used in an office where it could be in constant use from 9-5, or will it be at home where it’s main use will be 7am on weekdays and 9am at the weekend?
- Now compare them with Test Cases based on Test Classes: www.tulon.cz/QA/kettle-revisited (from page 2 onwards)
- The difference in coverage is only result of experience with Test Analysis (which you will gain with time), „subconscious mental FMEA“... **And of using Test Classes!**

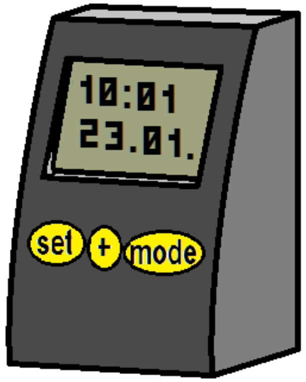
Finite state machines are your friend

State Transition Testing, Stateful and Environmental Testing

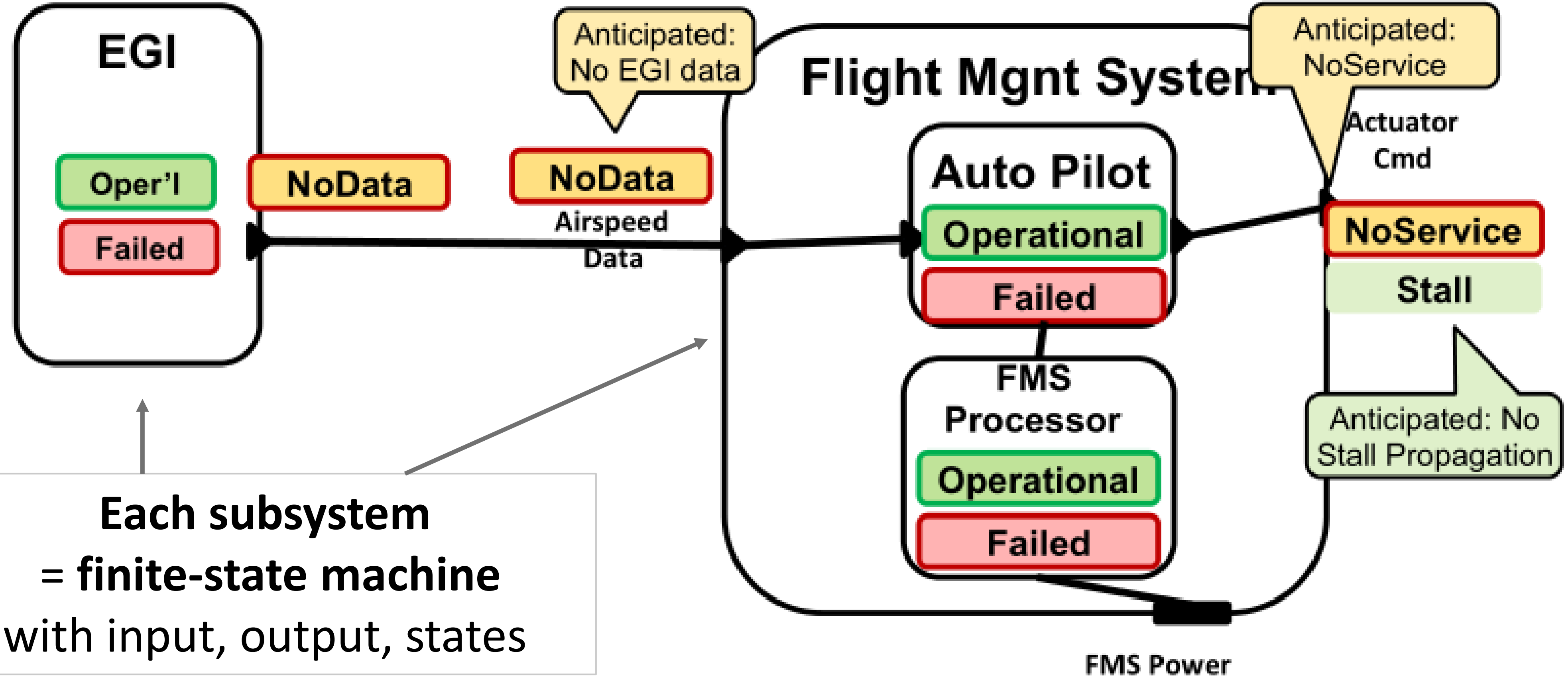
State transition testing

- When the SUT transitions between discrete states:
  - Unexpected „from-to“ transition could trigger Defect
  - Long sequences of state transitions could trigger Defect
  - Repeated „oscillations“ between states could trigger Defect
- Solution: State Transition testing based on State Transition diagrams, specifying:
  - Model of states SUT could occupy
  - Transitions between the states
  - Events which trigger the transitions
  - Results of the transitions
- Watch for
  - data flows – biggest source of defects in STT!
  - unfounded assumptions of transition results (*transition fails...*)
  - unfounded assumptions of expected events (*2-button-press...*)
  - combination of state transitions with outside factors (*velocity...*)





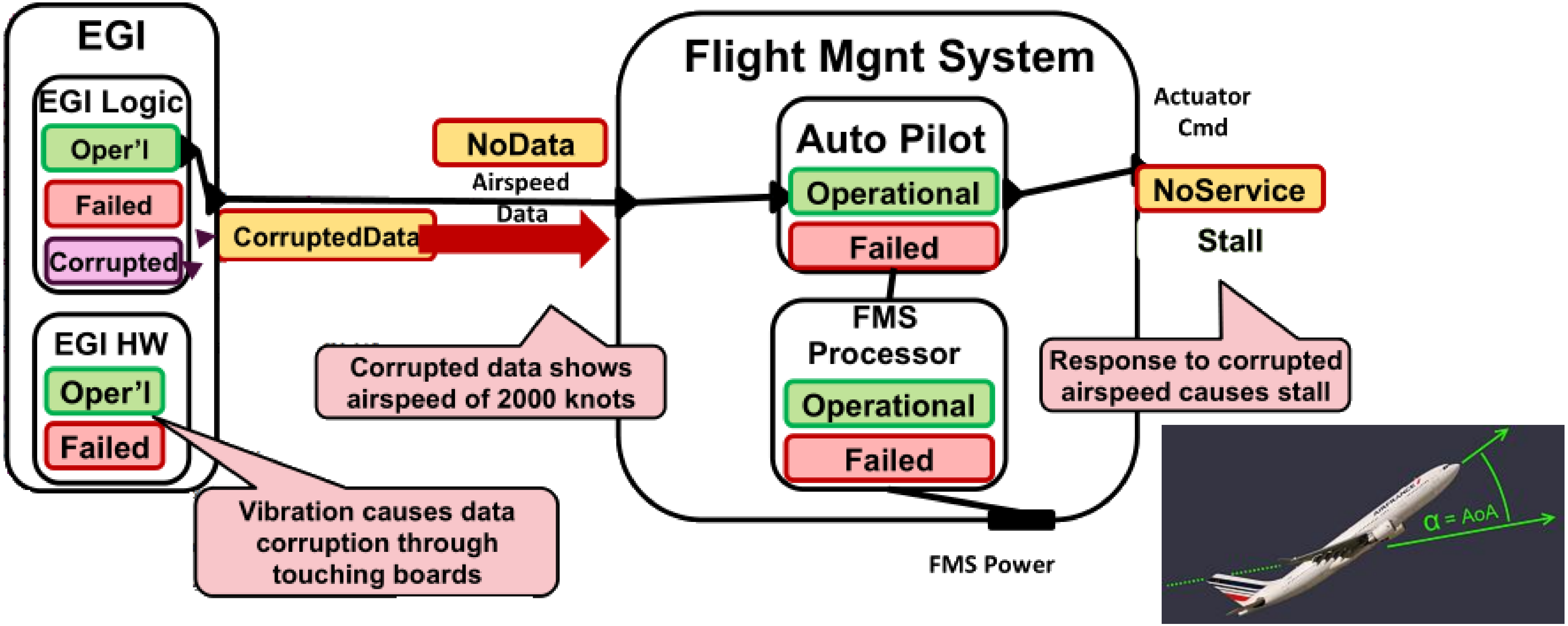
Stateful I/O analysis: what was assumed („EGI could have 2 states“)



Each subsystem
= finite-state machine
with input, output, states

Graphics: Carneige Mellon University, Safety modelling with AADL, 2015, under fair-use

Stateful I/O analysis: when assumption is wrong („unsanitized 3rd state of EGI“)



Vibration causes data corruption through touching boards

Corrupted data shows airspeed of 2000 knots

Response to corrupted airspeed causes stall

Graphics: Carneige Mellon University, Safety modelling with AADL, 2015, under fair-use

Function Lists

Function List

- All the Acceptance Criterias are Explicit Functions. Eg.:
  - User should be able to authenticate
- But this requires many pre-requisite or supporting Implicit Functions.
- And each implicit function could require more Implicit functions. Eg.:
  - Write Login screen GUI
  - Create Credentials storage
    - Create and deploy database engine configuration file
    - Create ODBC connection to the credentials database
    - Handle exceptions regarding Credentials storage
      - Issue error messages and document them
  - Add function Function to reset password
- **Without Function List, Developer would solve these informally as-they-pop-up**
- **And unforseen Implicit functions would require Refactoring = source of bugs**

Expert methods for finding Failure Modes and Nodes

Failure Mode Effect Analysis and Fault Tree Analysis

Finding unassumed problems: Failure Mode Effect Analysis & Fault Tree analysis

- „Failure mode“ = „How things fail“
- „Effect analysis“ = „What happens if they fail“
- „Root cause“ = „beginning of cascade of failure“
- „FMEA“ is analysis to identify and record:
  - all discrete functions/components which could fail,
  - **how** could they fail
  - **what** would happend if they fail,
  - what could be done to make them **not fail**
- FTA is analysis to identify and record:
  - all the root causes which could cause a given failure
  - **why** can a chain of events leading to failure get triggered
  - **what** functions, systems, modules or environmental variables could facilitate a failure
  - what failures which are harmless on their own could cause disaster when **combined** with others
  - what is the **probability** that such failures would occur and combine to worsen resulting effects

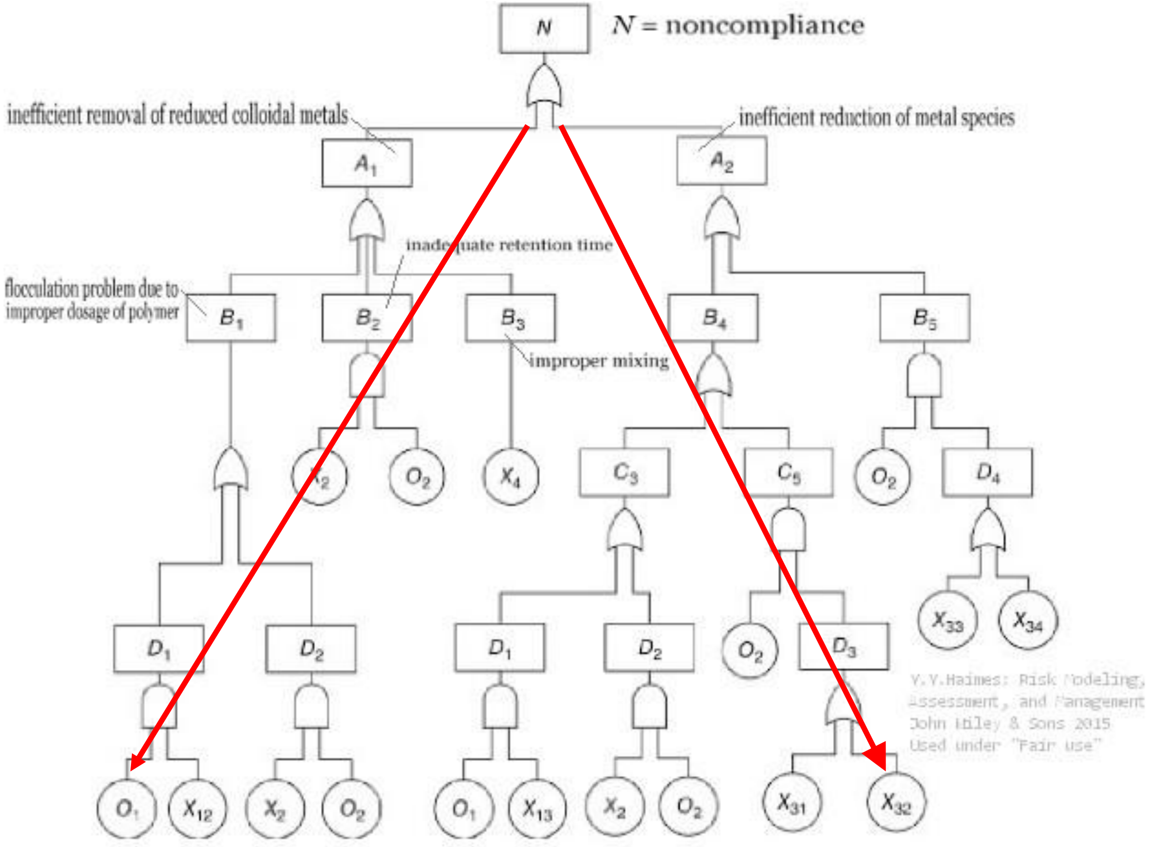
System	TV	Flight tests
Polaris AX	1958	🟥🟥🟥🟥🟥🟩🟩🟥🟥🟩🟥🟥
LGM-30A	1959	🟥🟩🟩🟩🟩🟩🟩🟩🟩🟩🟥🟩
Polaris A1	1959	🟩🟥🟩🟩🟥🟥🟩🟩
SM-65E	1960	🟥🟥🟥🟩🟥🟥🟩🟩🟥🟥
LGM-30B	1962	🟩🟩🟥🟩🟩🟩🟩🟥🟩🟩🟩🟩🟩🟥
LGM-30F	1964	🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩🟩

FTA VS physical flight test failures ($1,4M/$11,5M)
Guess which system used FTA and which didn‘t?

FMEA vs. FTA: similar but opposites

FTA:

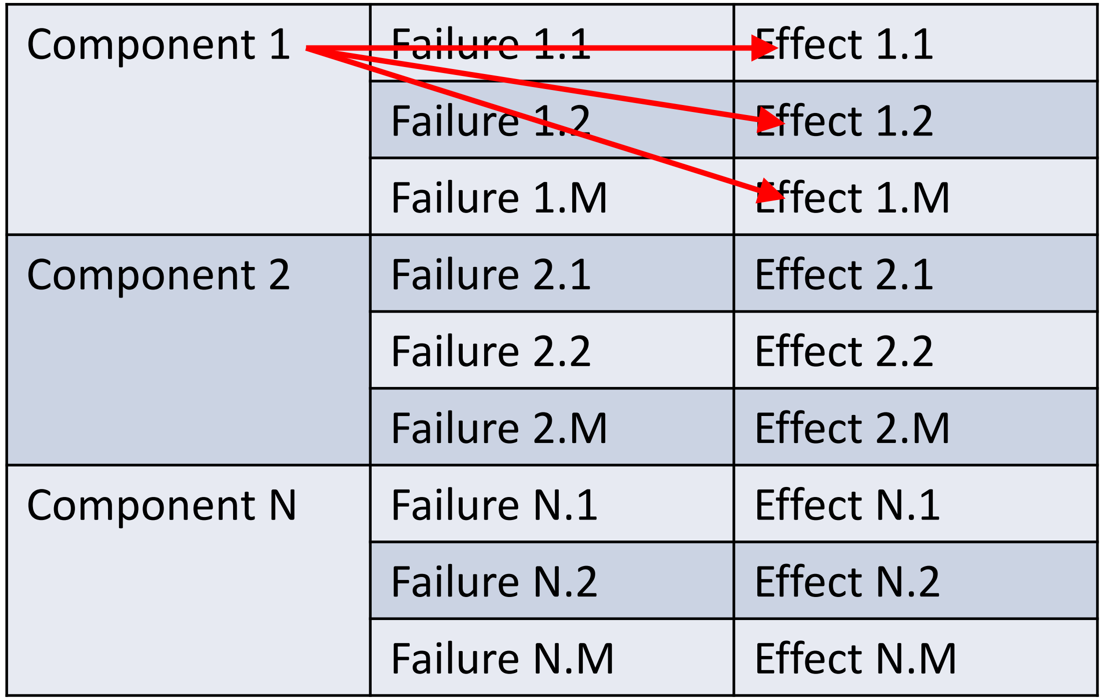
deductive - identifying a major failure, then finding possible causes. **Focus: failure reasons**.


N
N = noncompliance
inefficient removal of reduced colloidal metals
inefficient reduction of metal species
A1
A2
inadequate retention time
flocculation problem due to improper dosage of polymer
B1
B2
B3
B4
B5
improper mixing
X2
O2
X4
C3
C5
O2
D4
D1
D2
D1
D2
O2
D3
X33
X34
O1
X12
X2
O2
O1
X13
X2
O2
X31
X32

Y.Y.Haimes: Risk Modeling, Assessment, and Management John Wiley & Sons 2015 Used under "Fair use"

FMEA:

inductive - identifying components and listing their possible failures. **Focus: failure effects**.



Component 1	Failure 1.1	Effect 1.1
	Failure 1.2	Effect 1.2
	Failure 1.M	Effect 1.M
Component 2	Failure 2.1	Effect 2.1
	Failure 2.2	Effect 2.2
	Failure 2.M	Effect 2.M
Component N	Failure N.1	Effect N.1
	Failure N.2	Effect N.2
	Failure N.M	Effect N.M

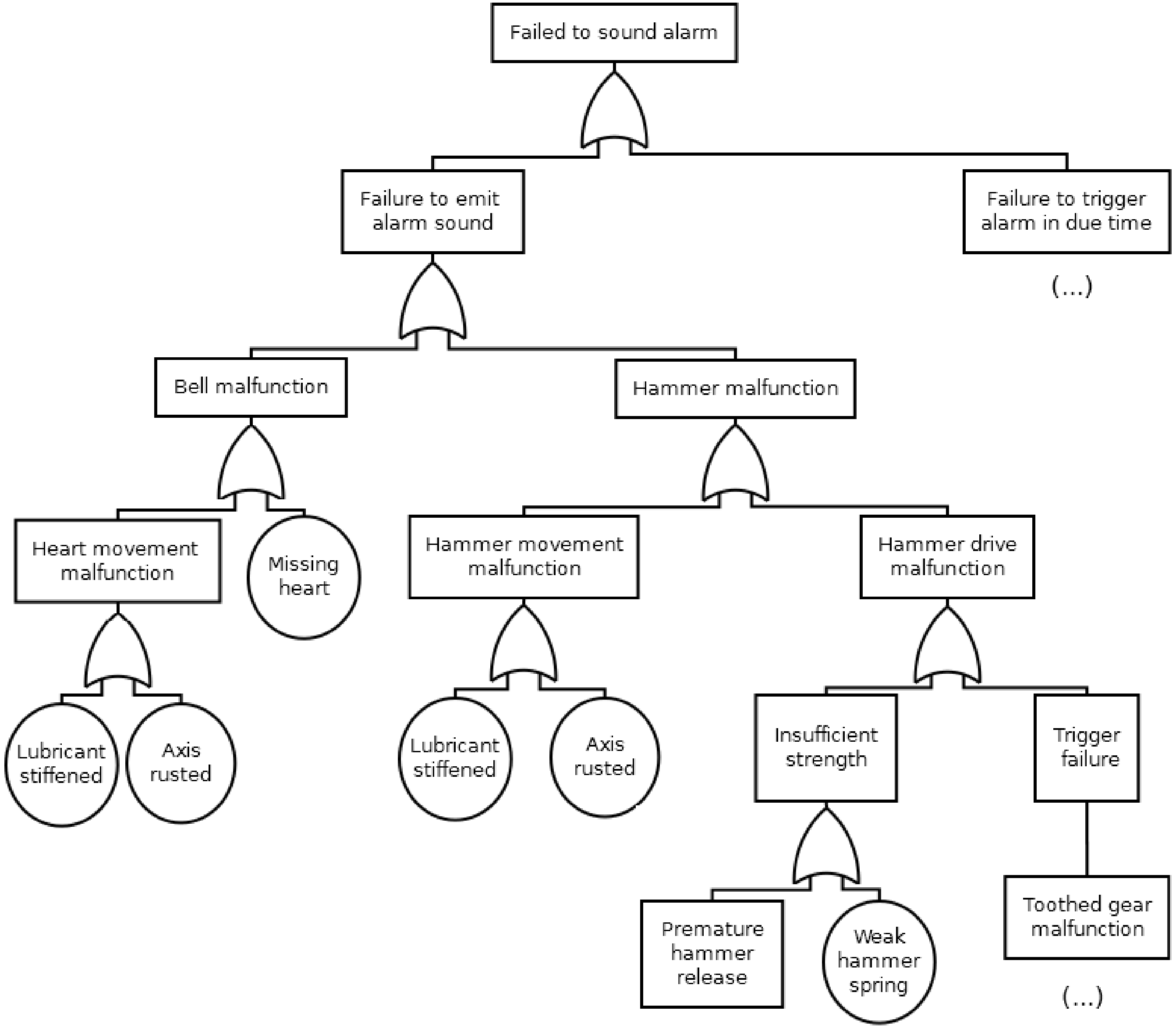
FTA: Traditional contents

- Traditional FTA encodes following information
  1. **What is the fault we want to prevent/diagnose („Top-level event“)**
  2. **Can this fault consist of failure on a lower level, or is it lowest possible root cause?**
  3. **Or could it happen because of *combination* of failures on lower level(s)?**
     a) **what are the logical relations – AND, OR, or AND with external influence?**
  4. **Is such a fault realistic? No hunches/prejudice – reason why yes/no**
  5. **Could this fault *also be caused by something else*?**
- FTA can show *how errors/defects propagate* within the system
- FTA is like preventive „RCA“ (before, not after incident happens)
- **FTA can show single-point-of-failures which need to get special QA attention**

FTA: how to do it

- **Step 1: set up definitions**
  1. What is the **top event** (loss of system OR loss of mission)
  2. What is the **scope of SUT being analyzed** (version, initial states, inputs)
  3. What is the **resolution/boundaries** (when we stop asking „why“)
- **Step 2: for each step**
  1. Identify **ALL possible causes**, not only obvious
  2. Identify relation of causes: independent = OR / dependent = AND / dependent in sequence = Priority AND
  3. Consider only nearest immediate causes: „Think small“, **smallest possible steps, do NOT jump** ahead to root cause – <u>this is crucial for FTA</u>!
  4. Consider **Primary fault** (=failed because defect), **Secondary fault** (=failed because unexpected environment/state), **Command fault** (=worked as designed but triggered in incorrect moment)
  5. Decide if the step is **victim** of other cause „upstream“ or **primary cause** ○

FTA simplified demo: alarm clock (without probabilities)

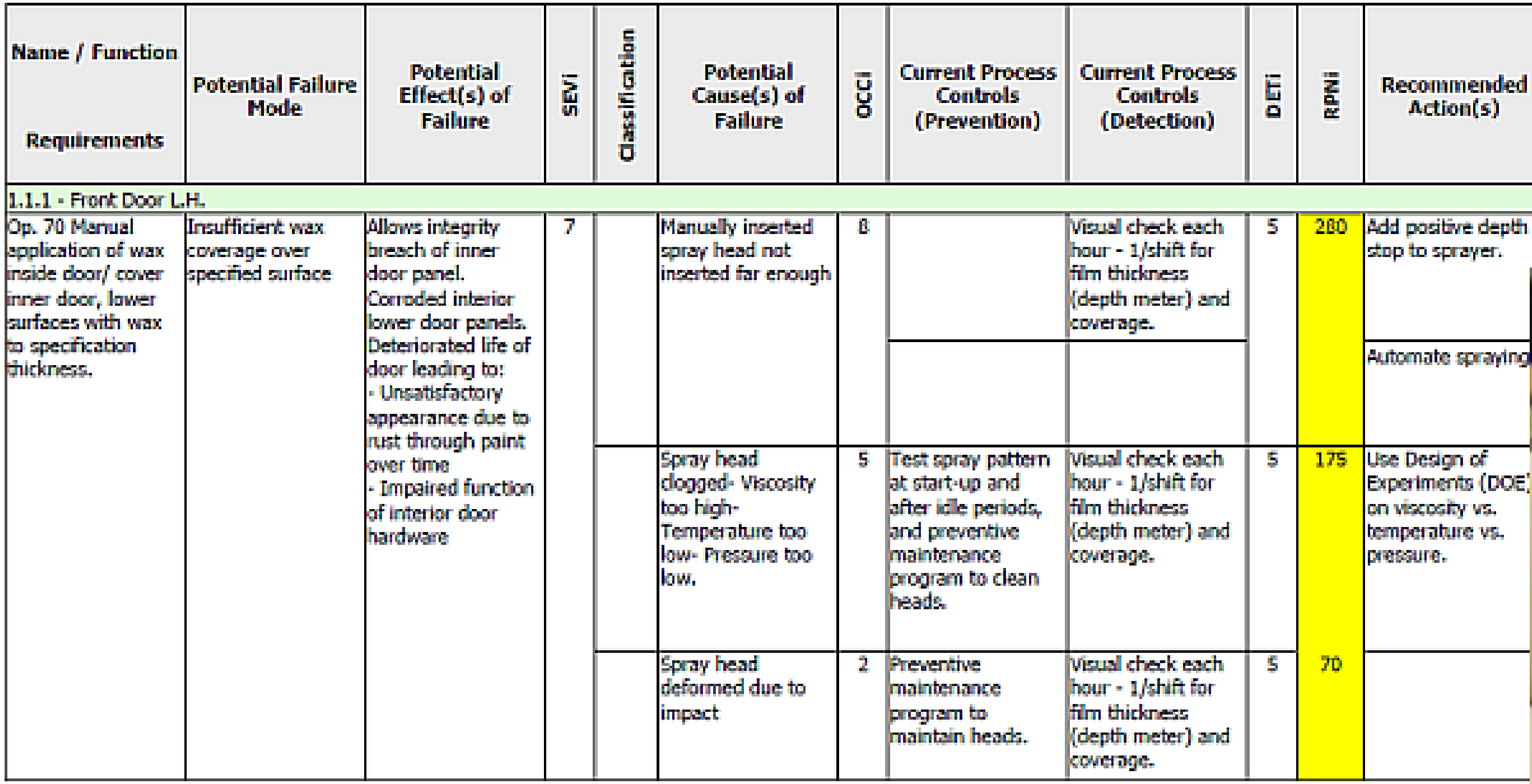


FMEA: Traditional contents

- Traditional FMEA contains information
  1. **How the components/subsystems fail** (=not who coded defect, but how did the defect get triggered)
  2. **What went wrong** once the component failed (=chain of subsequent errors and failures)
  3. **What areas were impacted** by the chain of failures
  4. **What is the severity**, **probability** of occurence, and likelyhood of **detection** by current/planned tests
- It then uses the *(Severity) x (Occurence) x (Detection)* to calculate „Risk Priority“
- „Risk Priority“ → where to focus extra tests, attention and development effort
- **Important: data about real-life accidents and failures are recorded to FMEA to serve as inputs-to-consider in future developments and tests**
- If you start on lowest possible component level, you identify root causes
- Boundaries are needed (FMEA on every nut-and-bolt level unrealistic)

FMEA: Traditional contents VS. „lightweight custom“

This is how traditional industrial FMEA looks like:



Name / Function Requirements	Potential Failure Mode	Potential Effect(s) of Failure	SEVi	Classification	Potential Cause(s) of Failure	OCCi	Current Process Controls (Prevention)	Current Process Controls (Detection)	DETi	RPNi	Recommended Action(s)	Responsibility & Planned Completion Date	Action Results				
													Actions Taken & Actual Completion Date	SEVr	OCCr	DETr	RPNr
1.1.1 - Front Door L.H.																	
Op. 70 Manual application of wax inside door/ cover inner door, lower surfaces with wax to specification thickness.	Insufficient wax coverage over specified surface	Allows integrity breach of inner door panel. Corroded interior lower door panels. Deteriorated life of door leading to: - Unsatisfactory appearance due to rust through paint over time - Impaired function of interior door hardware	7		Manually inserted spray head not inserted far enough	8		Visual check each hour - 1/shift for film thickness (depth meter) and coverage.	5	280	Add positive depth stop to sprayer.	Mfg Engrg - 3/10/2003	Stop added, sprayer checked on line.	7	2	5	70
											Automate spraying						
					Spray head clogged- Viscosity too high- Temperature too low- Pressure too low.	5	Test spray pattern at start-up and after idle periods, and preventive maintenance program to clean heads.	Visual check each hour - 1/shift for film thickness (depth meter) and coverage.	5	175	Use Design of Experiments (DOE on viscosity vs. temperature vs. pressure.						
					Spray head deformed due to impact	2	Preventive maintenance program to maintain heads.	Visual check each hour - 1/shift for film thickness (depth meter) and coverage.	5	70							

FMEA: Traditional contents VS. „lightweight custom“

For commercial SW, let‘s select just the columns we absolutely cannot go without

Function / Component	Potential Failure Mode	Potential Failure Effect	Sev	Potential Causes of Failure	Current Detection Controls	Recom. Actions
What should it do	What could fail	What will happen if it fails	##	Defect/Failure 1	Test 1	Improvement
				Defect/Failure 2	Test 2	Improvement 2

SAE J-1739 Severity categories

FMEA severity value (SAE J-1739): good consideration

Rank	Criteria of effect severity	Effect
1	No effect	None
2	Fit & finish/estetic problem, noticed only by discriminating customer (<25%)	Very minor
3	Fit & finish/estetic problem, noticed by average customer (50% of customers)	Minor
4	Fit & finish/estetic problem, noticed by most customers (>75%)	Very low
5	System operable, reduced performance of non-mission-critical feature	Low
6	System operable, inoperable non-mission-critical feature	Moderate
7	System operable but with reduced performance / customer dissatisfied	High
8	System inoperable: loss of primary function / **mission loss**	Very high
9	**Threat to safe system operation, data,** or compliance, with warning	Hazardous+warning
10	**Threat to safe system operation, data,** or compliance, **without warning**	Hazardous w/o warning

Point: safety/compliance errors without warning/announcement are the worst

FMEA simplified demo: alarm clock

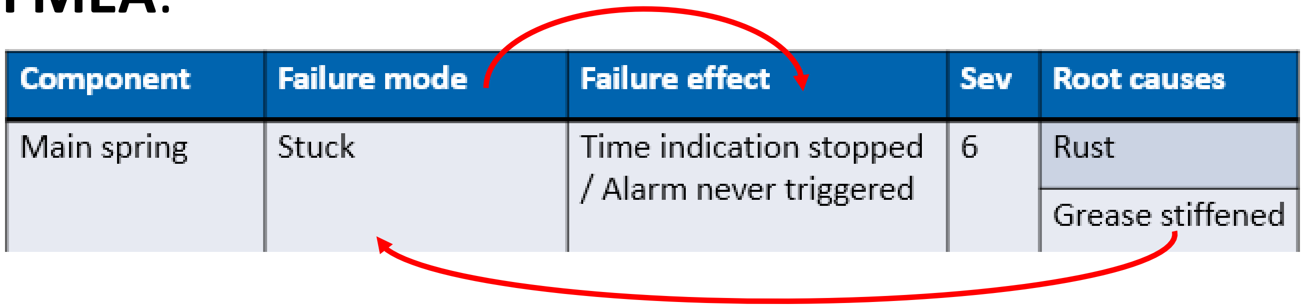
Fn/Component	Failure mode	Failure effect	Sev	Root causes
Clock	Stopped	Time indication stopped / Alarm never triggered	6	Rust
				Mech. stiffened
	Delayed	Time indication late / Alarm delayed	8	Int. friction
				Power loss
				Bad calibration
Alarm	Stuck acoustic mechanism	Alarm sound silenced	7	Int. friction
				Mfg defect
				Power loss
	Trigger stuck	Alarm never activated	10	Mfg defect

FMEA and FTA: pro‘s and con‘s

- FMEA
  - Good **if you know the functions/components** of the system, their weak points (program code...)
  - **Less likely to miss possible failure mode** – IF you have low-enough scope
  - Inherently suggests improvements and risk mitigation
  - Cannot „filter“ and ignore components on the same scope/level (1000 bolts = all or none)
  - Requires very systematic approach, time-consuming
  - Shallower in regards to possible root causes, concurrent causes
- FTA
  - Good **if you know how the system works** and could fail (history od incidents, domain knowledge...)
  - **Less likely to miss possible root cause** – IF you understand the system deep enough
  - Can discover one critical, vulnerable component amongst others (1 bolt in 1000)
  - Can discover how errors propagate within system
  - Can discover how concurrent errors combine and affect each other and system
  - Inherently allows calculation of overall failure risk given all failure nodes failure probabilities
  - Easily overwhelmed with dependencies/interference accross failure nodes
  - Invalid assumptions could cause omitting critical failure mode/node
  - Exhaustive: FTA scope is not entire system or function, but single failure mode

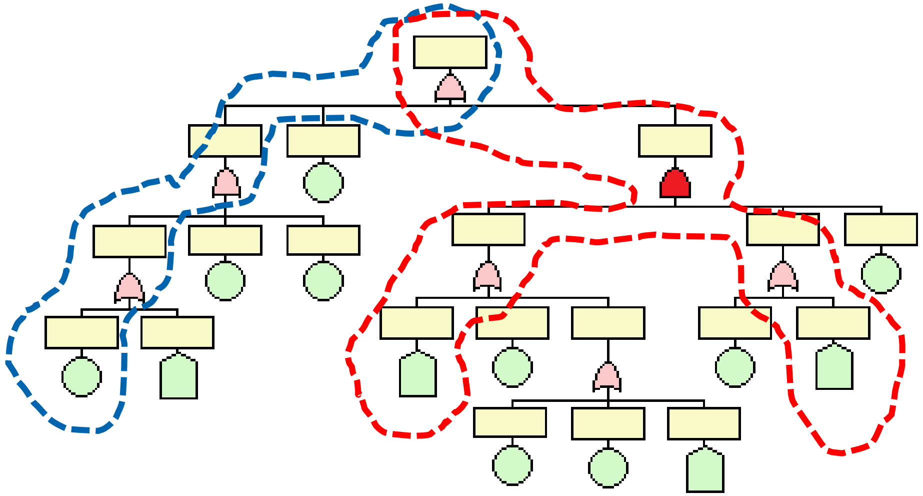
FMEA and FTA: how to use them to find Test Cases

- FMEA and FTA: „Wheen *<root cause>* happens, SUT *<fails>* because *<Failure mode>*“
- Root cause = Test input; SUT behavior = Expected result
- → Test Case: „Even if *<root Cause> happens*, SUT should *handle* it*“
- **FMEA**:



Component	Failure mode	Failure effect	Sev	Root causes
Main spring	Stuck	Time indication stopped / Alarm never triggered	6	Rust
				Grease stiffened

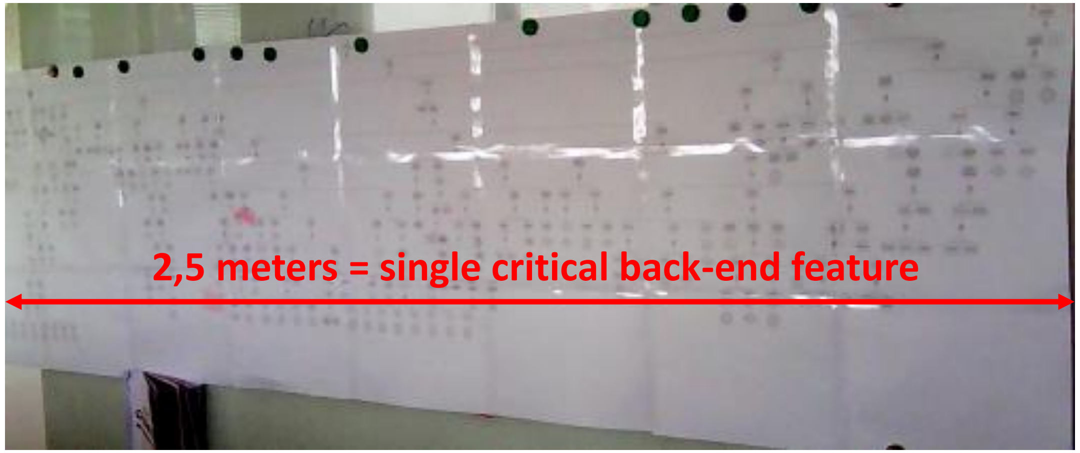
- **FTA**:
  - „Cut sets“ = minimal sufficient combination of RCs
  - Cut set = 1 path through ORs but all through ANDs
  - Cut sets could convert to Test Cases...
  - By „running“ each Cut Set and evaluating results



*If the SUT isn‘t fail-safe, failure might not be preventable – but it is sanitizable: error messages, exception handling...

FTA real-life observations

- FTA is „holistic“ tool, contributing 40% to QA but 60% to Devs
- FTA is better by order of magnitude when pair of QA+DEV does it
- Software-based FTA enables quick refactoring (freeware: OpenFTA)
- But you lose track of nodes „beyond the screen edge“ → physical FTA printout is a must
- FTA expansion of RCA of historical defects is best tool to make trully good new regression tests
- FTA enabled us to find potential serious defects which surprised even senior DEVs

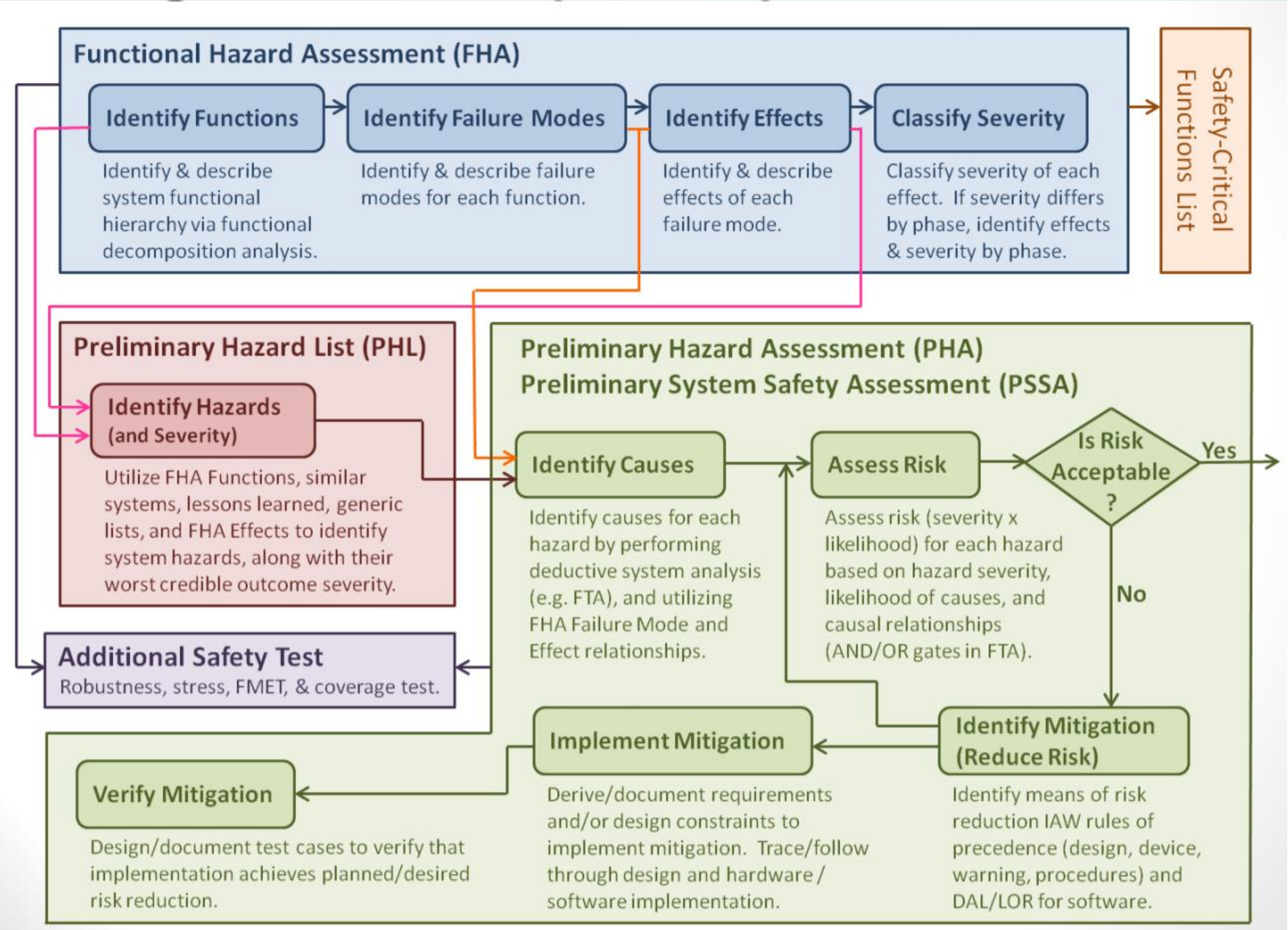

2,5 meters = single critical back-end feature

Self-study: „FTA reminders“

How it all combines together

Design&Analysis phases, FMEA, FTA, Fail-safe as a workflow

Aerospace-level QA: DO-178C (FAA mandatory SW QA)



Functional Hazard Assessment (FHA)

Identify Functions

Identify & describe system functional hierarchy via functional decomposition analysis.

Identify Failure Modes

Identify & describe failure modes for each function.

Identify Effects

Identify & describe effects of each failure mode.

Classify Severity

Classify severity of each effect. If severity differs by phase, identify effects & severity by phase.

Safety-Critical Functions List

Preliminary Hazard List (PHL)

Identify Hazards (and Severity)

Utilize FHA Functions, similar systems, lessons learned, generic lists, and FHA Effects to identify system hazards, along with their worst credible outcome severity.

Preliminary Hazard Assessment (PHA)
Preliminary System Safety Assessment (PSSA)

Identify Causes

Identify causes for each hazard by performing deductive system analysis (e.g. FTA), and utilizing FHA Failure Mode and Effect relationships.

Assess Risk

Assess risk (severity x likelihood) for each hazard based on hazard severity, likelihood of causes, and causal relationships (AND/OR gates in FTA).

Is Risk Acceptable ?

Yes

No

Identify Mitigation (Reduce Risk)

Identify means of risk reduction IAW rules of precedence (design, device, warning, procedures) and DAL/LOR for software.

Implement Mitigation

Derive/document requirements and/or design constraints to implement mitigation. Trace/follow through design and hardware / software implementation.

Verify Mitigation

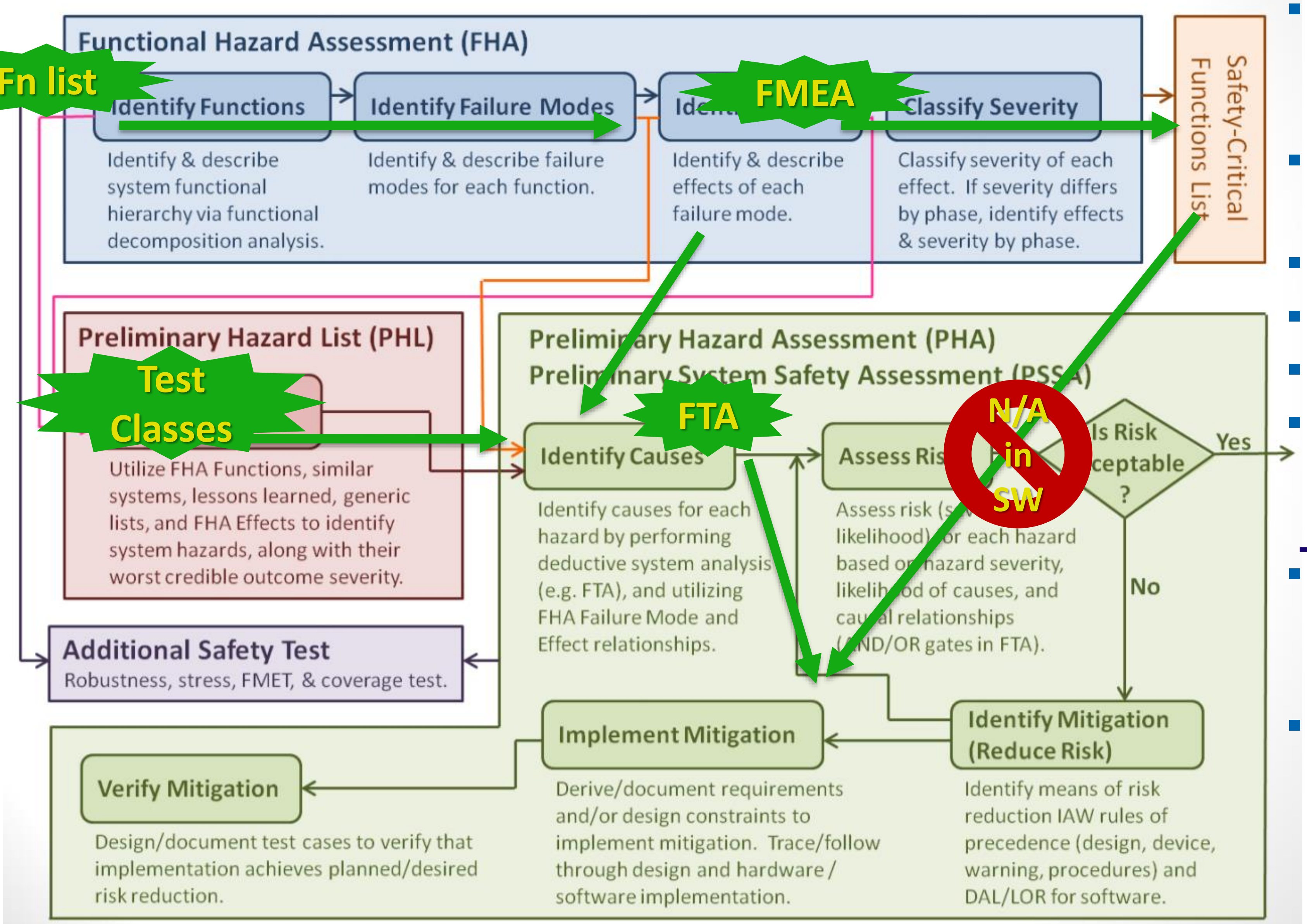
Design/document test cases to verify that implementation achieves planned/desired risk reduction.

Additional Safety Test

Robustness, stress, FMET, & coverage test.

(So... How comes
B737MAX MCAS?!)

Aerospace-level QA, hacked for non-certified SW for time/effort savings



Functional Hazard Assessment (FHA)

Identify Functions

Identify & describe system functional hierarchy via functional decomposition analysis.

Identify Failure Modes

Identify & describe failure modes for each function.

Ide

Identify & describe effects of each failure mode.

Classify Severity

Classify severity of each effect. If severity differs by phase, identify effects & severity by phase.

Safety-Critical Functions List

Preliminary Hazard List (PHL)

Utilize FHA Functions, similar systems, lessons learned, generic lists, and FHA Effects to identify system hazards, along with their worst credible outcome severity.

Additional Safety Test

Robustness, stress, FMET, & coverage test.

Preliminary Hazard Assessment (PHA)

Preliminary System Safety Assessment (PSSA)

Identify Causes

Identify causes for each hazard by performing deductive system analysis (e.g. FTA), and utilizing FHA Failure Mode and Effect relationships.

Assess Ris

Assess risk (s likelihood) for each hazard based on hazard severity, likelihood of causes, and causal relationships (AND/OR gates in FTA).

Is Risk ceptable ?

Yes

No

Identify Mitigation (Reduce Risk)

Identify means of risk reduction IAW rules of precedence (design, device, warning, procedures) and DAL/LOR for software.

Implement Mitigation

Derive/document requirements and/or design constraints to implement mitigation. Trace/follow through design and hardware / software implementation.

Verify Mitigation

Design/document test cases to verify that implementation achieves planned/desired risk reduction.

- Determine basic SW desired functions: system+subsystem level -> Fn List
- Use FMEA to find how desired SW functions could fail -> Severity
- FMEA severity -> Critical Fn List
- Use FTA to find CrFnL root causes
- Use TCls to consider ENV risks
- For each dangerous and possible failure identified by FTA, design-in protections, mitigations

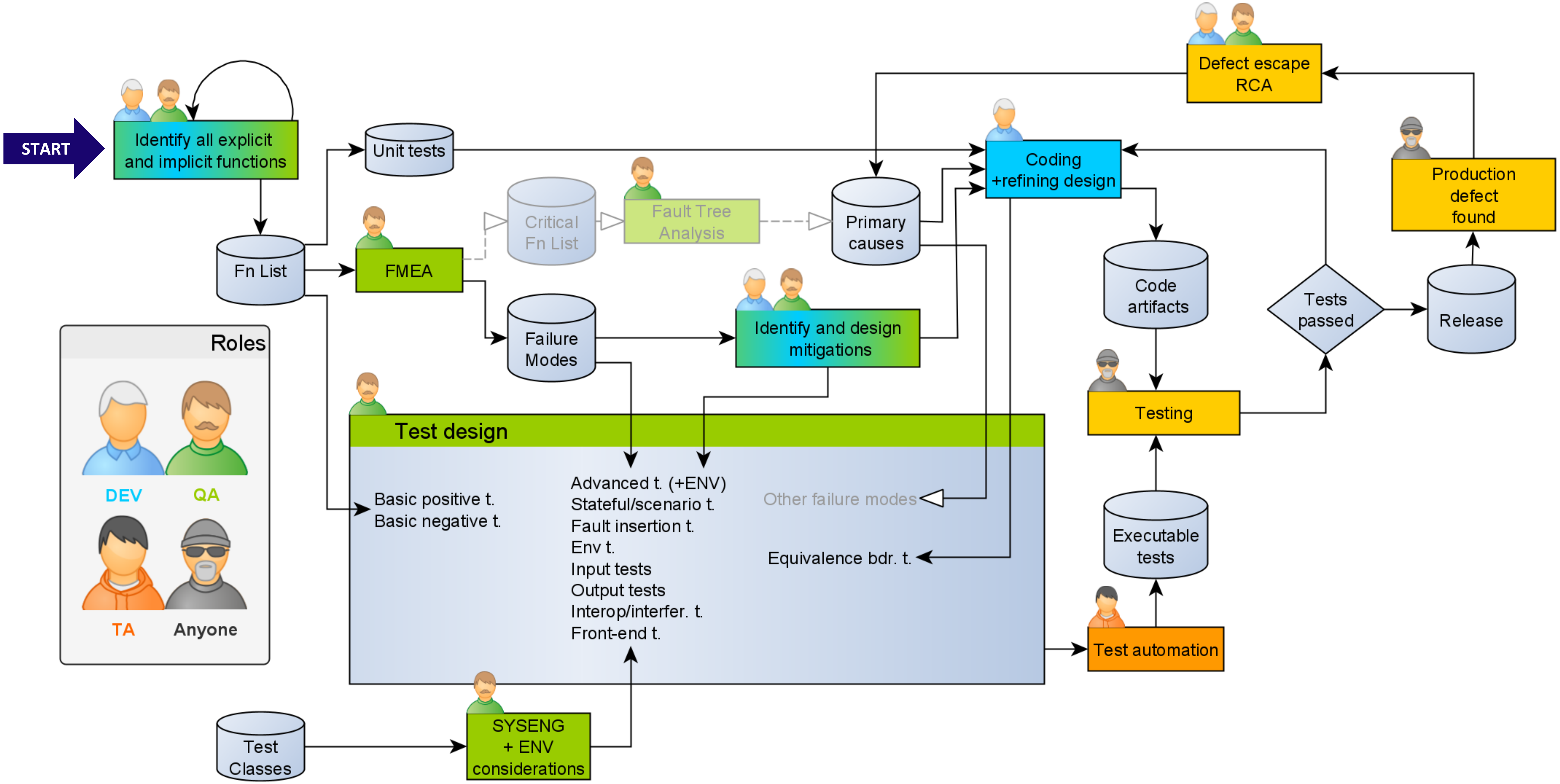
- start coding & designing tests to cover these design req's, including Fault Insertion Tests
- design tests to cover lifecycle quality – „*ability testing“

architecture, analysis, design← | → implementation

„Think first, code later“

ca technologies

Complete QA+Safety+Reliability workflow for non-certified SWDEV



Recommended resources

- **Increasing QA maturity level:**
  - B.Beizer – „Black-box testing“ ($), „SW testing techniques“ ($)
  - ISTQB Foundation level
  - RTCA DO-178C ($) / EUROCAE ED-12C ($), DO-248C
  - Joint Software Systems Safety Engineering Handbook
- **Test analysis specification-based techniques (EP, BVA, EP DI, STT, TCl):**
  - ISTQB „Advanced Test Analyst“, „Advanced Technical Test Analyst“ courses and Syllabi
  - BS 7925-2 „Standard for SW Component testing”, ISO/IEC/IEEE 29119-4:2015 „Test techniques” ($)
  - IEEE 829:2008 „Standard for SW testing documentation” ($)
- **Test analysis failure/risk-based techniques (FMEA, FTA, RCA):**
  - C.Wilhelmsen, L.T.Ostrom – „Risk Assessment: Tools, Techniques, and Their Applications” ($)
  - C.S.Carlson – „Effective FMEAs” ($)
  - MIL-STD-1629A „Procedures for performing FMEA”
  - NUREG-0492 „Fault Tree Handbook”
  - NASA Fault Tree Handbook with Aerospace Applications
  - DOE-HDBK-1208-2012 Volume 1 „Accident and Operational Safety Analysis Techniques”
  - TOR-2014-02202 „RCA Best Practices Guide” ($)
- **Domain-free resources:** https://broadcom.box.com/v/ATAT-QA-materials

Wrap up & final workshop

Recap and put to use what you have learned

Recap: what we learned about Quality Assurance?

- **If you‘re QA Engineer = all-in-one Tester + Test Analyst + Reviewer, then**
- **You need to cover:**
  - positive, negative, destructive, functional and integration testing
  - checking for Developer‘s invalid assumptions, for unforseen conditions and operating states
  - Boundary Values Analysis (+related tests)
  - Real or „subconscious mental“ FMEA or FTA = analysis of potential failures and their root causes
    - hypothetical, discovered by analysis of components/systems/functions
    - empirical, recorded from the bugs/failures observed „in the wild“ at customer defects etc.
- **You decide what‘s important using**
  - Severity-Probatility 2D chart
  - Consequence integrity levels
- **You should use Test classes** (*designated groups of Test Cases)* not to forget anything

How to start Quality Assurance instead of QC?

- Think not only how the feature/code should work/succeed, but how it could fail
- Find team member with talent for Test Analysis and add TA to development cycle
- Use Fn lists, Fn-level FMEA to gather Critical failures list; factor-in ENV worst-case sc.
- If failures can cause damage of customer (data), use system modelling (Fault Trees or at least Data Flow Testing) → test cases + sanitization in code
- New features: architect-in maintenability, extensibility, scalability
- Always identify system boundaries /envelope and verify reliable function via Stress tests
- Always do reviews with feedback on all of:
  - Architecture, x-ability, and feature/product design
  - code
  - test cases
- When customer defect occurs, do at least informal RCA:
  - why did the defect get in (NOT „blame game“, but „how to avoid similar next time“)
  - why didn‘t the architecture contain it, why didn‘t code fail-safe it
  - why din‘t existing tests catch it, what tests to add to catch future similars

Q&A

And further resources

Final workshop: create Test Cases using the advanced methods

- In teams of two, write High-level Test Cases: not of an electric kettle, but this time, the test subject is **e-mail system**

Requirements:

- Single-user, single-platform thick client for desktop
- Send e-mails via external SMTP server
- Receive e-mails via external POP3 server
- Store, retrieve, read, search already sent and received e-mails

- Use HLTCs, no need for detailed LLTCs
- Point: hands-on try all techniques covered:
  1. FHA (collective effort) + FMEA + Integrity levels + Severity levels
  2. Test Classes (design just 2 test cases per each Test Class and move on)
  3. FTA of critical failure mode of your choice (trace at least 2 Cut Sets to their Primary Causes ○)
  4. Bonus: suggest architecture/implementation mitigation of possible critical failure modes

Fn List → FMEA → Critical Fn List → FTA + ENV via Test Classes → Mitigation