The last Knowledge transfer: 12 years of advanced QA

Ondrej Tulach

Intro, whoami, sylabus

- Programmer in C#.NET, IBM PL/I, C; Sysadmin/Infra; QC→QA→Safety Engineering
- 2011: CA Panvalet / Librarian
  - Test Automation fails when Test Coverage is poor
- 2012: CA ACF2
  - First forays into QA feedback for „future-proofing“ the product
  - ISTQB training by RCBS & certification
- 2013: CA Db2Tools – Recovery Analyzer, Log Analyzer
  - Fixing the mess and coverage gaps with Test Classes
  - Learned BS 7925 test techniques – State Transition testing, Data Flow testing etc.
- 2018: achieved seniority
  - Learned RCA, FMECA and applied FTA for the first time
  - Systems Thinking
- 2020: XCOM
  - Started applying Function Lists + FMECA (Fn Hazard Assesment) with great success

Software Quality Control

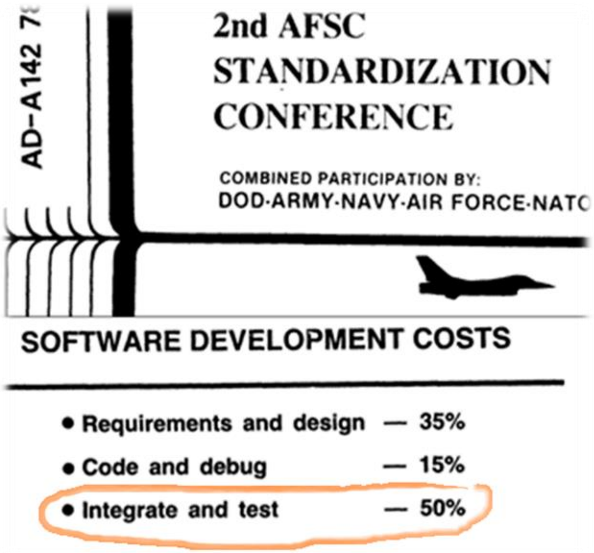
BROADCOM®

Basic Positive Testing, TDD and Unit tests

- So-called “shift-left testing” using TDD and UT is *not* QC/QA testing nor substitute
- TDD’s Unit tests are not really tests – they’re TODO: list of the Developer
  - Yes, they are proven to make developers write much better code
  - But they are Basic Positive Tests verifying just that what the Developer already catched
  - Anything the Developer didn’t expect could break the program...
  - And even system integrated from perfect Units could be riddled with integration errors
  - Detailed explainer: TDD and Unit tests: the neuroscience behind and the fundamental limitations
- Quality Control needs to go way beyond Basic Positive Testing
  - Verify all edge cases and boundaries: Boundary Values Testing, Equivalence Partitions
  - Verify unexpected inputs handling: input testing, Human Operator Error Testing
  - Verify unexpected state transitions: State Transition Testing of back and forth, out-of-sequence
  - Verify load limits: Stress Testing
- Simply put, a lot of Destructive, Edge/Boundary and Input handling tests
  - Usually more than 2× to 4× tests than the Basic Positive ones

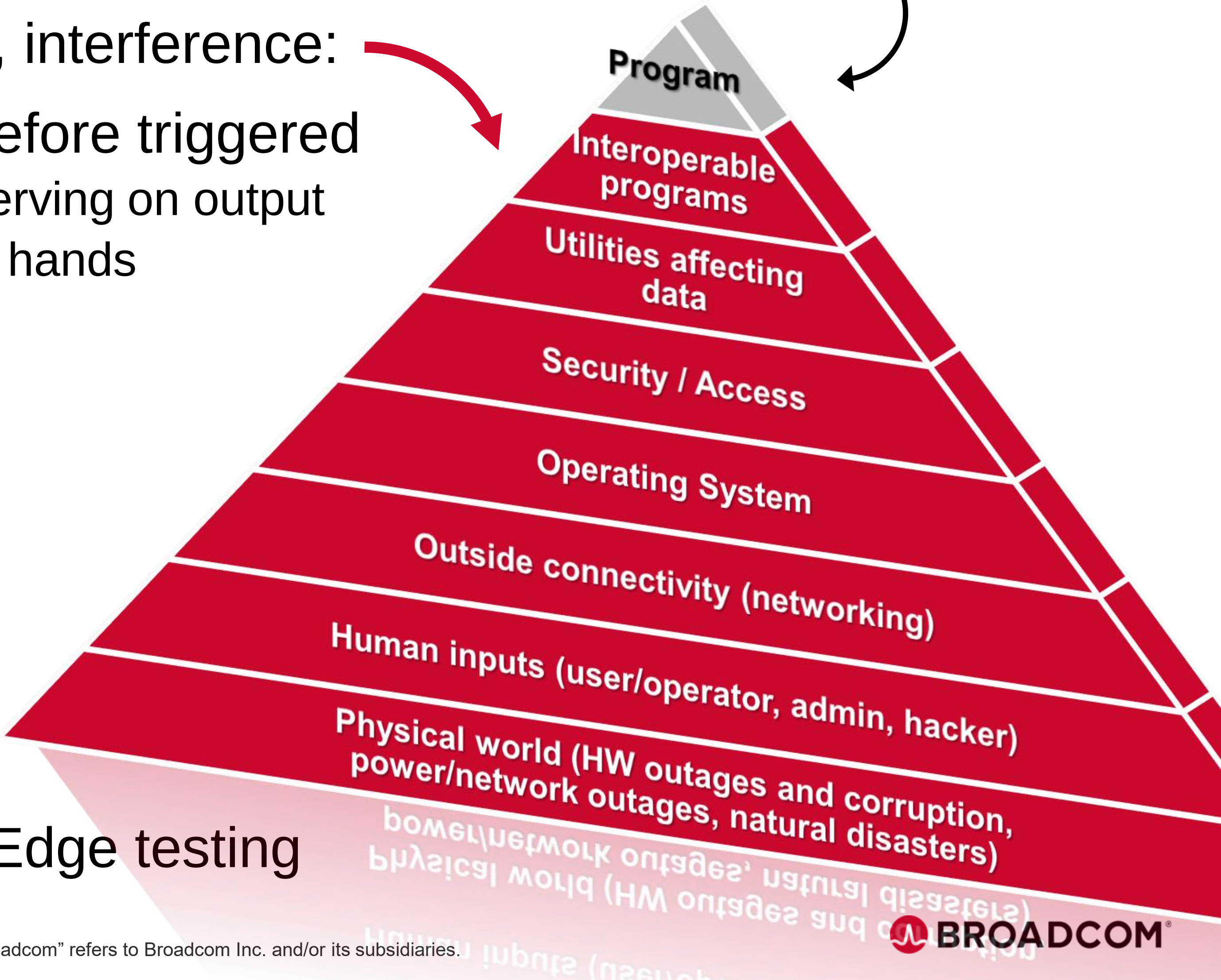
Data Flow Testing: why development is faster than testing

- Up to 50% of defects is caused by data of unexpected/unhandled values, not code
- Each variable in a program code can be
  - P-variable affecting code flow,
  - C-variable providing useful calculation (payload/output),
  - Used as both
- If a developer changes just 1x code line with P-variable affected...
- QC needs to verify impact on *every single LOC downstream* where P-var used!
  - Cannot stress the ramifications of this enough...
  - Extreme unbalance between minor DEV code changes and QC effort
  - Extremely robust Automated Regression tests the only solution
    - But must be *so* robust to really catch *every* P-variable use!
- This used to be understood in the big SW industry in the past:
  - Managers: now you know the secret



Real test coverage: foreseeing the externalities

- Beginner SW engineer fallacy: focus on idea of isolated Program alone
- IRL, many layers of of dependencies, interference:
- Preventive QA must deal with risks before triggered
  - Interoperability survival on input and preserving on output
  - Resiliency to utilities changing data under hands
  - Handling of unauthorized access/action
  - Compatibility with OS (and updates)
  - If networked: handling *whatever comes*
  - Human Operator Error Testing (massive!)
  - Thinking about data corruption (FIT)
  - Handling unexpected states (outages during critical phase, recovery)
- How? **Test Classes Cheatsheets**
- Yes, this is *on top* of the Destructive/Edge testing



The Test Classes Cheatsheet QC revolution

- Reaction to failure: Automated Dropped Object Recovery (ADOR) testing
  - BIG effort, unprecedented tests nr, many duplicities... And still glaring Test Coverage Gaps!
  - Chaotic: object types testing, scenarios testing, parameters testing... On one huge heap
  - Hard to reverse-find a test case (“Did we test this? IDK, have to read the entire doc”)
- Search for remedy
- Test Classes (re)discovered in IEEE829, **Test Classes Cheat Sheet** v1 created
- 2016: first successful use on Quite Point Analysis
  - Remedy for all Coverage Gaps identified in ADOR testing – user inputs, utilities... ✓
  - ~120 TCs neatly organized in 11 Test Classes ✓
  - More test cases, minimized Coverage Gaps ✓
  - No duplicates, logical hierarchical organization, easy reverse-lookup ✓
- Test Classes Cheatsheet template = baseline for new QC work
  - Continually updated whenever Testing Coverage Gap found => introducing QC self-repair loops!

Test Classes

- Test Class Cheatsheet sources / templates:
  - One-page generic TCCS: https://tulon.cz/TCCS
  - Product-specific TCCS templates confidential to DB2BKR.PLA, CORE.XCOM
  - My personal detailed TCCS template: https://github.com/tuloncz/TestClassesCheatSheets/
    - When I learn about gaps in coverage worth cheatsheeting, I add it here and retrofit to work ones
- How to start?
  - Read Test Classes explainer on CAWIKI: https://bsg-confluence.broadcom.net/x/0YzXHQ
  - Ask PLA or XCOM team for their product-specific TCCS templates
  - Or download my TCCS template from Github – it's now licensed as CC-BY, OK to use commercially confidentially without sharing it
- Iterate through the template and FOREACH(testclass *Tcl* in TCCS_template):
  - Ask yourself: *"can this Tcl be relevant to our code/feature/product, or is it Non/Applicable?"*
    - IF Tcl.relevant, write as many underlying test cases needed to satisfy Test Coverage for given Tcl
    - ELSE set Tcl = "N/A"
- When all relevant Test Cases are written, prioritize with DEV and PO

Optimizing the Test Classes for efficiency and cycle time

- Extra thorough testing can cause tensions with Cycle Time-optimizing managers
- Rational optimization can save 400% time on Test Design with Test Classes
  1. High-Level Test Cases instead of LLTCs: no expected results, no test steps
  2. Shift test case feasibility testing from Test Design to the Testing phase

- Risk-Driven Testing can save anything in the Testing phase
  - Test Classes-driven test design will create *much more* tests than ad-hoc design (2× to 10× more)
  - But not all tests *need* to be executed / automated
  - Risk-Driven Testing:
    1. Consult with developer to assign Severity rating to test cases
    2. Consult with PO / manager to balance Severity Cut-off threshold
    3. The remaining test cases are Accepted and Managed Risk (known untested)
    4. Based on the Severity Cut-off Threshold, any % of testing time can be saved
  - Important mindset: QA task is to inform PO/FM about risks.
    PO/FM manages the risks by allocating resources.

 |

BROADCOM

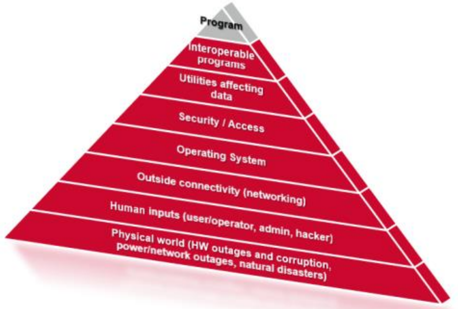
Quality Assurance

QA mindset revolution: the “defective washing machine” example

- A parable which changed my perception and goals
- Customer‘s washing machine fails. Warranty repair. RCA using 5Why‘s:
  1. Machine failed because motor jammed
  2. Motor jammed because engine ball bearing lubricant leaked
  3. Lubricant leaked because incorrect O-ring size installed
  4. Incorrect O-ring installed because mistake of the assembly workers
  5. Mistake because O-ring sizes too similar, impossible to notice by-eye

conqelectric.com, under Section 107 Fair Use of the Copyright Act

- QC: add new test case to measure O-ring sizes in completed motors
- QA: redesign the motor shaft/O-ring combo so it‘s impossible to install wrong size!
- Profound implications: QA to address the entire QC Risks Pyramid preventively?!
  - Before: FSS, DDS – killed by Agile
  - Now: Lean Function Lists – compatible with Agile



Bad and good QC/QA reactions to a defect

- When the Root Cause is found, you can:
  1. **Blame and punish** the culprit => people will focus on hiding their errors instead of fixing them
  2. Set-up **rigid processes** to disallow it the next time => “Death by ISO” == bad QA
  3. Suggest product/process **design changes** to prevent the error while retaining simplicity

- Practically, in the IT world:
  – Example of #1: git-blame used not as a joke
  – Example of #2: lengthy Coding standards, mandatory checklists (i.e. cannot skip N/A items)
  – Example of #3: preliminary SW function and hazards analysis and mitigations

- How do you do the analysis and mitigations?
  – **Systems Engineering**: mock-ups how will the future customer use SW, ID of subsystem & iface
  – **Function Lists**: discovering the untold implicit requirements and data structures to avoid hacks

BROADCOM

Software Systems Thinking

Systems thinking: an universally applied abstraction

- In detail, all projects and systems look very different
- If you „zoom out“, they all share identical traits:
  - They consist of multiple discrete components integrated to a system
  - They have interfaces and/or connections between those components
  - They have inputs
  - They have outputs
  - Phenomena can *propagate* between components trough interfaces (commands, data, errors)
- Reliable design, implementation and testing needs to cover „all bases“
  - Inputs
  - Outputs
  - Components function
  - Interfaces
  - Propagation of desirable AND undesirable phenomena
  - Integration of components to a system

BROADCOM

Systems thinking: a simple matter of mindset

- Logically decompose the whole to subsystems based on their function

Purpose
- Transfer power from human legs to wheels

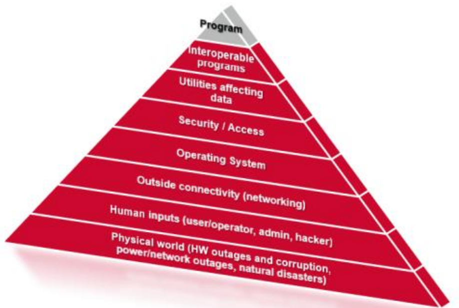
Components
- Pedals
- Cranks
- Sprocket
- Chain drive

Properties
- Sprocket ratio
- Leverage length

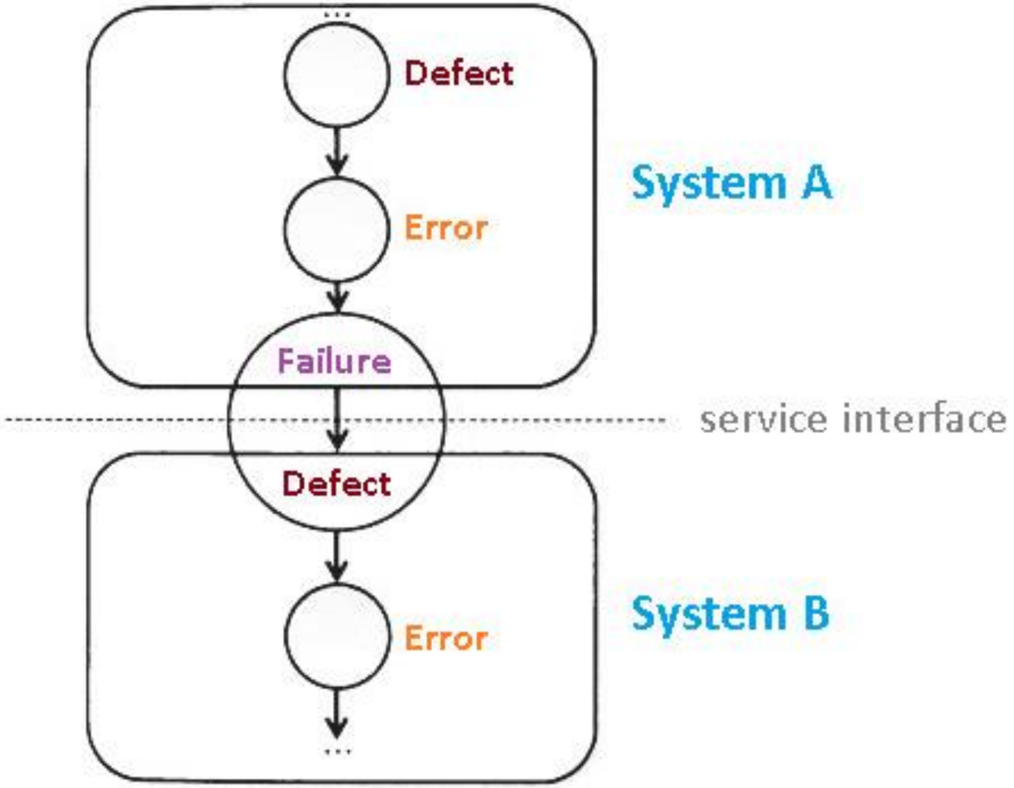
Limitations
- Torque resistance
- RPM before overheat

SYSTHINK interfaces translated to QA

- A system is integrated from components connected through interfaces
- In software bussines, we always have these interfaces:
  1. Between units of code (functions, classes, objects, instances)
     – Unit testing
  2. Between modules and features (DLLs etc.)
     – Integration and System testing
  3. Between external inputs and the system processing them
     – Sanitizing inputs – almost always under-tested & under-coded
  4. Between the system and the human operator
     – Everything from intuitive operation, UI and UX, through output clarity, to error messages
  5. Between our software and other software
     – „Interoperability“ – customers can have many SW product „chained“ together via their I/O in cascade
- Flows across interfaces must be analyzed, tested, controlled
- The pyramid of program dependencies again – but different focus



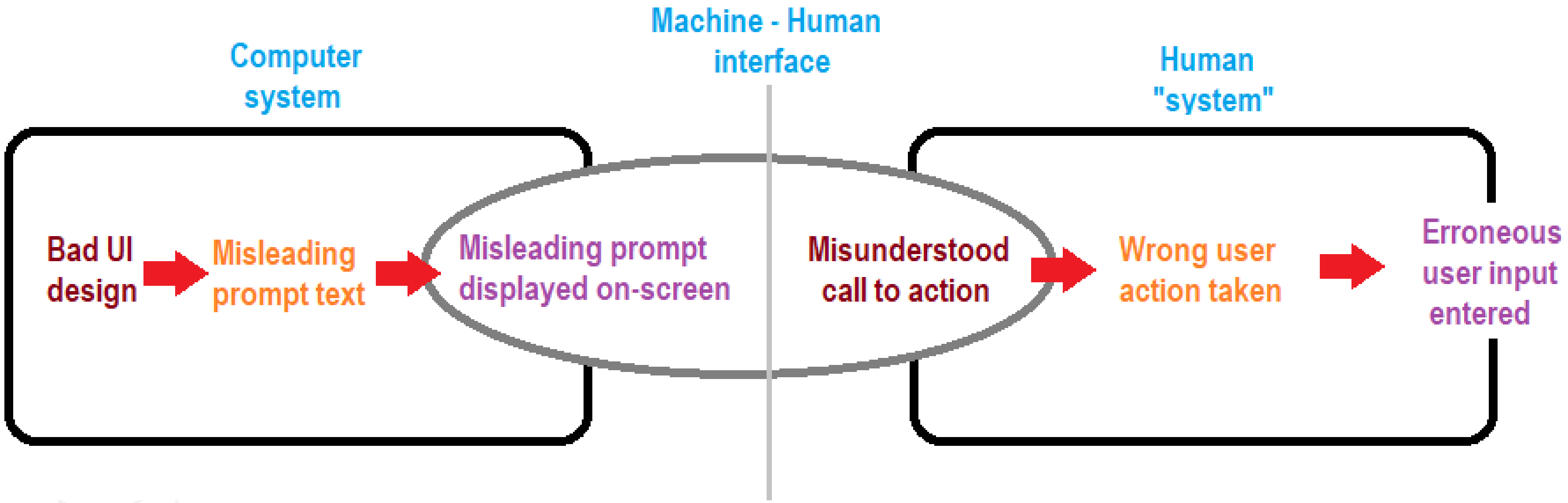
Propagation of errors trough systems and interfaces



- Since we now see systems and interfaces...
- We can model precise causality of errors:
  - **Defect** is the true root cause
  - **Error** is the consequence of defect being triggered
  - **Failure** is the visible manifestation of the error (usually on some interface)
- Consequences observed:
  1. Errors can *propagate* through multiple interfaces and systems = each next system a "victim"
  2. The failure you observe != defect. RCA needed!
  3. If true RCA not discovered, the "fix" can hide the true defect, making it more dangerous
- Preventive QA: consider each interface a possible source of Input errors!
- RCA: "5 Whys" to swim upstream

“The User Component” of a software system

- Favorite industry excuse: „user error“ => end of story, case closed
- Systems thinking: human “system” is just a component of the integrated system
- Error propagation theory: user error can be victim of error-chain “upstream”

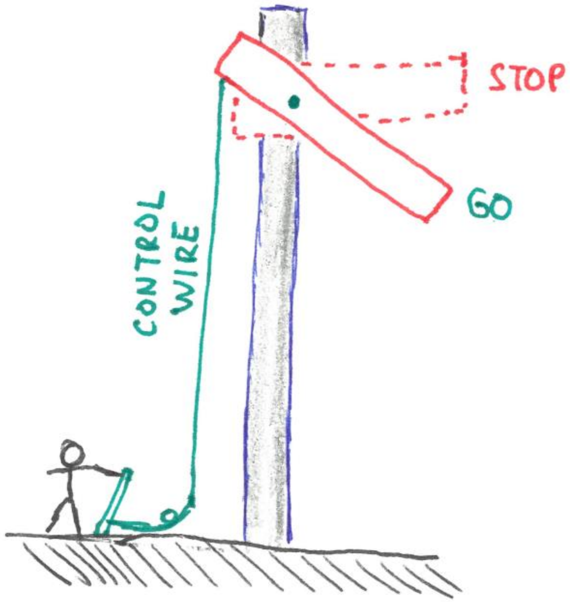


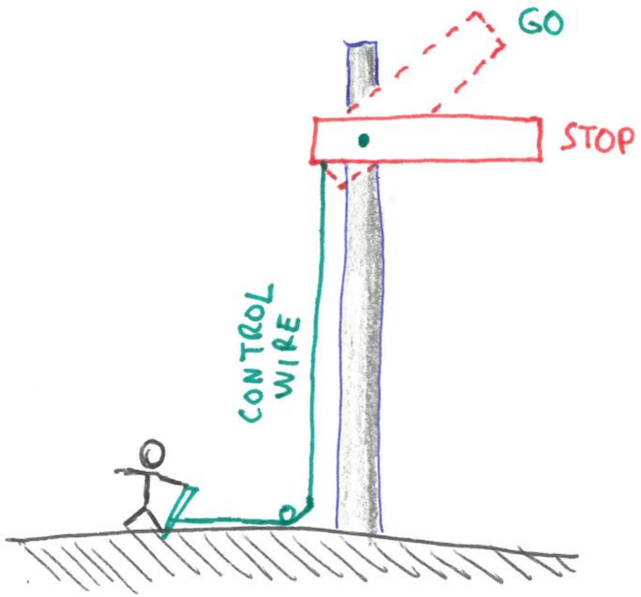
- QC vs. QA mindset revolution: is erroneous user a Root Cause - or a Victim?
  – If victim, preventive QA changes the system to eliminate Human Operator Errors!

Brief Intro to System Safety & Software Safety

BROADCOM®

Single point of Failure vs. Fail-safe





- If the wire breaks:
  - system *defaults* to GO
  - Fail-Unsafe
  - Risk: system loss (trains collide)

- If the wire breaks:
  - system defaults to STOP
  - Fail-Safe
  - Risk: mission loss (trains stopped)

BROADCOM

SPoF and Fail-safe in software

- **SW Single Point of Failure**
  – Actually two flavors

1. The failure crashes entire program
   – SEGFAULTs
   – Dereferencing pointer to invalid address
   – Your own memory/variables overlay
2. The failure causes unrecoverable mission loss
   – Endless loops
   – Waiting until unwritable dataset becomes writable without offering cancel & different choice
   – Throw error and abort... Upon condition which will never be satisfied
   – Non-crashing Overflow ruins calculation

- **SW Fail-safe**
  – Three conditions:
    – Detect error without crashing
    – Isolate error from causing damage
    – Retry or revert to backup
- Examples:
  – Validate inputs and provide guidance to remedy if invalid
  – Watchdog/timeout on resource access, then ask to abort or choose different
  – Detect impeding overflow, stop execution before, throw a clear error
  – Declare-initialize pointers with NULL, always check if pointer NULL before dereferencing
  – Watchdog loops:

```
int watchdog = LOOP_LIMIT;
while (current_line != '\0' && --watchdog > 0)
{
    ...
}
```


Safety is (often) the enemy Reliability: system loss v. mission loss

- Watchdog loop example:
  - If (watchdog limit < required loop iterations)
  - Then loop will not finish properly, some data will not be read/processed
  - Output will be erroneous = mission loss
- Competing imperatives: prioritize mission success VS. prioritize system safety
  - Truism: any safety mechanism will inevitably fail with false-positives
  - Fail-safe + False-positive detection of failure = preventive system shutdown = mission loss
- Tough choice: what is bigger risk?
  - Endless loop => SEGFAULT (safety), or incomplete data read => incorrect result?
- Answer: it depends
  - On the severity of failure of given system / subsystem / unit
  - But how can you tell?

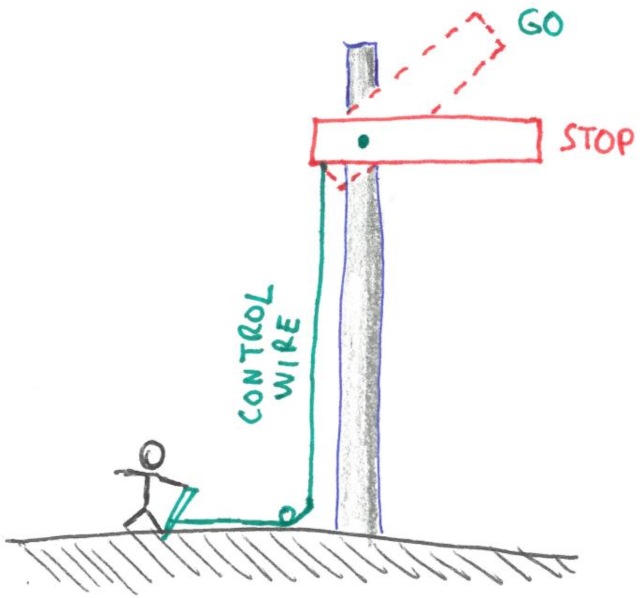
BROADCOM

The FMECA standard failure / severity table

Severity	Criteria	
1	No effect	No problem
2	Aesthetic problem (typos, GUI alignment...)	Noticed by < 25 %
3		Noticed by ~50 %
4		Noticed by > 75 %
5	Secondary / non-mission-critical feature	Bad performance
6		Inoperable
7	Primary function	Bad performance
8		Inoperable
9	Threat to system safety, data integrity, compliance	With warning
10		Without warning

Failure Mode Effect Criticality Analysis example

- For each component / subsystem
  - Identify **how** it could fail (Failure Mode)
  - What would **happen** then (Failure Effect)
  - And **how bad** would that be (Criticality)



- Semaphore consists of:
  - Pole, hinge, arm, wire+wire guides, controlling operator

- Caveat: components can fail in > 1 way

Component	Failure Mode	Effect	Criticality
Pole	Break & fall	Semaphore undetected, train wouldn't notice signal	10
Joint	Stuck	Previous arm position retained despite command	10
Arm	Break & fall	Train would spot broken semaphore, stop	8
	Bent & blocked	Previous arm position retained despite command	10
Wire	Break	Semaphore arm gravity-falls to lower position	8
	Stuck	Uncommanded incorrect position of arm, without detection!	10
Operator	Doesn't set signal	Commanded incorrect position of arm, without detection!	10

- We'd miss majority of dangerous states!
  - Even for 5 components. How many has your SW?

FMECA in Software... A dead end?

- So you have your product's new feature to test.
- And you have the FMECA table template:

Component	Failure Mode	Effect	Criticality

- *Now what?* What do you fill in?
- You're developing just 1 feature...
  – So it's just 1 line of FMECA? That would be useless...

- Answer: Functional testing's **Function List**

Function Lists

Function list - why

FHA phase

- Grooming, Personas, Acceptance criteria: few „explicit“ primary functions
- Real product: needs additional „fill the blanks“ dependent functions
- Back-end implementation: needs many support functions, data objects

- Example:

Acceptance criteria

„Adam the user can authenticate using username and password, so that the account is secured“

Explicit functions

Function to display and proces username and password

Function to add new user credentials

User function to reset the password

Support functions

Securely fetch password from input and DB

Compare password validity

Hash+Salt function to encrypt new pwd

BROADCOM

Function lists - what

- Consequences of ambiguous / unspecific “Personas” requirements:
  - Without Function List, Developer would solve the dependencies informally as-they-pop-up
  - Lot of surprise Refactoring required = source of bugs
- The discovered dependencies could grow recursively, adding more on same level:
  A. Write Login screen GUI
  B. Create Credentials storage
     1. Create and deploy database engine configuration file
     2. Create ODBC connection to the credentials database
     3. Handle exceptions regarding Credentials storage
        I. Issue error messages and document them
- When Function List is done before development and testing:
  - Much cleaner architecture, code design ✓
  - Function List can convert into Mock prototype / CONOP to validate with Agile customer ✓
  - Function List seamlessly converts into Unit Tests, driving comprehensive TDD ✓
  - Function List drives QA Basic Positive Testing to achieve ~100% Code Coverage ✓

Function list - how

FHA phase

- DEV and QA sit together and
  1. Write down all known Primary functions (from Acceptance criteria etc.)
  2. Identify dependencies - Secondary functions for Primary functions
  3. Identify datasources – files, databases, streams, user inputs, APIs etc.
  4. Identify needed Support functions for all Primary and Secondary Functions and accesing datasources
- QA does preventive risk analysis and asks DEV about concerns
  - DEV clarifies or accepts need to code mitigations
  - In case of important disagreement, PO is consulted about the risk => Risk-Driven Development
- The process gets more efficient with practice
  - From multiple multi-hour sessions at the beginning
  - To just one one-hour session after few iterations (for small/medium features)
- The most **Shift-Left** practice possible!

*except late design changes

Q & A

BROADCOM®

connecting everything®