A MANAGER‘S PRACTICAL GUIDE TO SOFTWARE QUALITY CAVEATS

Ondrej Tulach

Senior SW engineer / SQA / SW reliability, 12 years in CA/AVGO

Throughout 12 years, I‘ve seen enough to learn hard lessons

- The good and the bad of Waterfall and Agile
- Manual and Automated regression testing
- Ad-hoc and systematic/data-driven test design
- Targeted and Random-driven testing
- Teams led by former engineers and non-engineers
- I developed test automation from scratch and resumed/repaired pre-existing auto
- Experienced 5 test frameworks – and saw phase-out cascade-delete *all associated test data with it! (HP QC/ALM, PTG1, Version1, etc.)*
- How metrics influence, steer, damage Quality efforts
- What has really worked to decrease Customer Defects over time

You can learn what will work and what will not – without repeating the mistakes

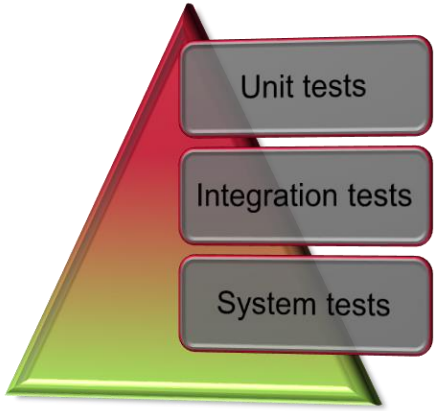
BROADCOM

Everyone only ever sees
The tip of the Iceberg

The testing words: 99% of people get them all wrong

- Testing is (subset of) Quality *Control*.
- Quality Control is **not** Quality *Assurance*.
- Quality Assurance is **not** Software Reliability Engineering.

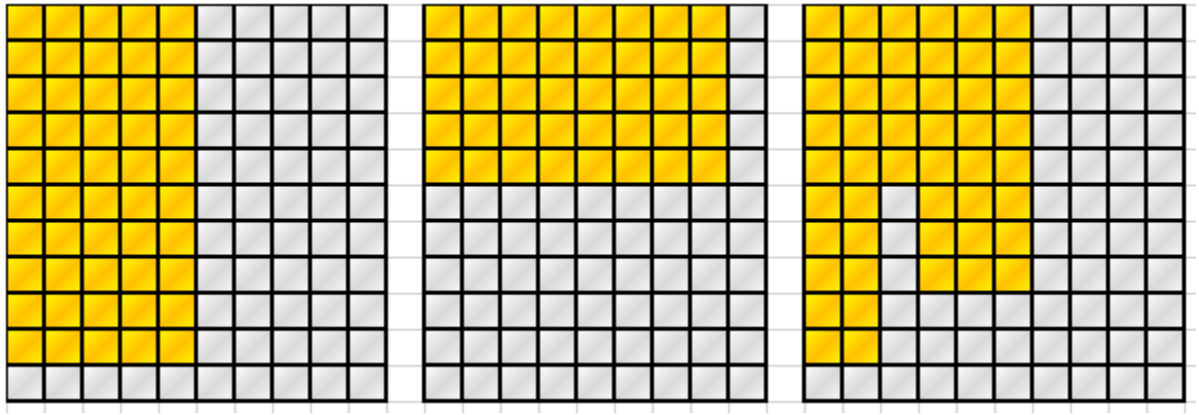
- Unit test is not –by far!– a System test.
- Smoke test is not –by far!– a Regresion test.



BROADCOM

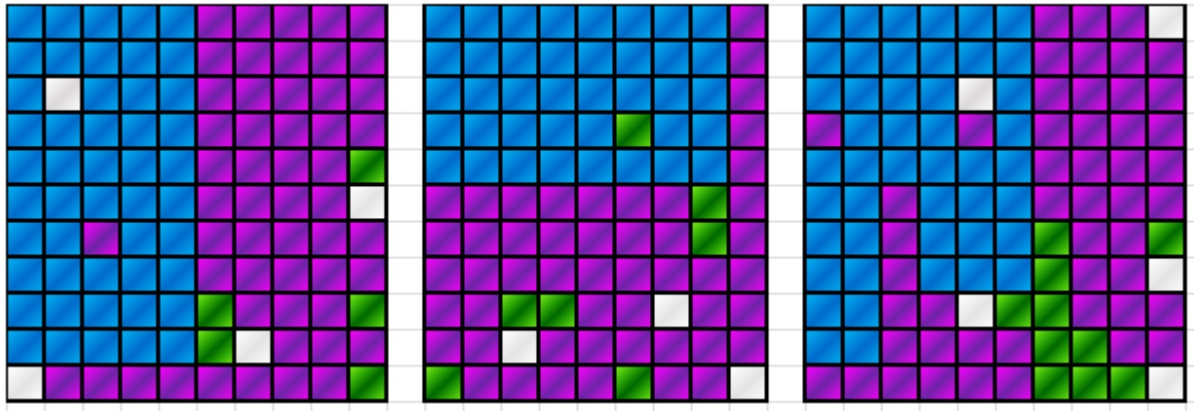
Unit testing vs. System Functional testing

- Unit tests / Code Coverage tests:



- Coverage:
  - The function as the Developer *intended* it
    - Expected inputs yield correct outputs
  - Equivalent to 100% Code Coverage
  - Doesn‘t cover the *missing code* needed to handle unexpected or diffucult scenarios

- Functional+integrated System tests:

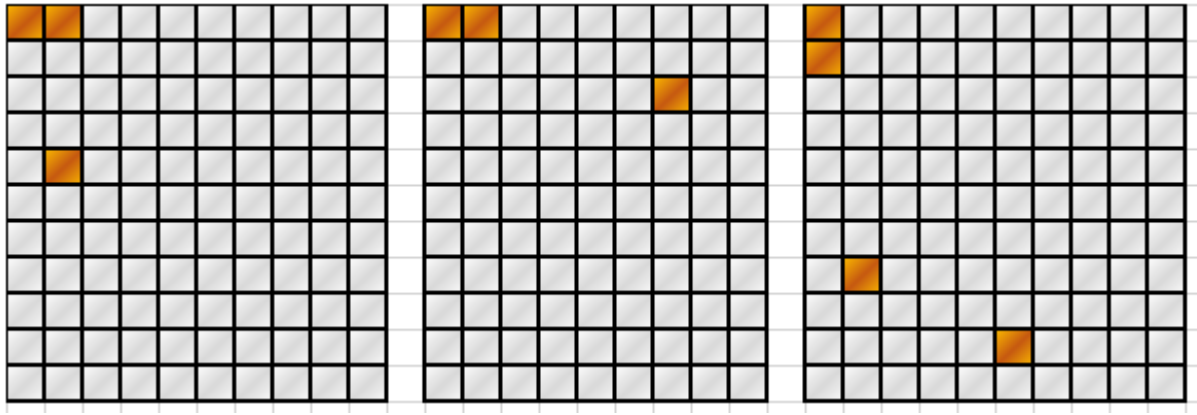


- Coverage:
  - *Independently* the same basic function
  - Hadling unexpected user inputs, data errors,
  - Handling unexpected states, sequences
  - Handling system influence (clock,proc,I/O...)
  - Product integration (shared config, data...)

BROADCOM

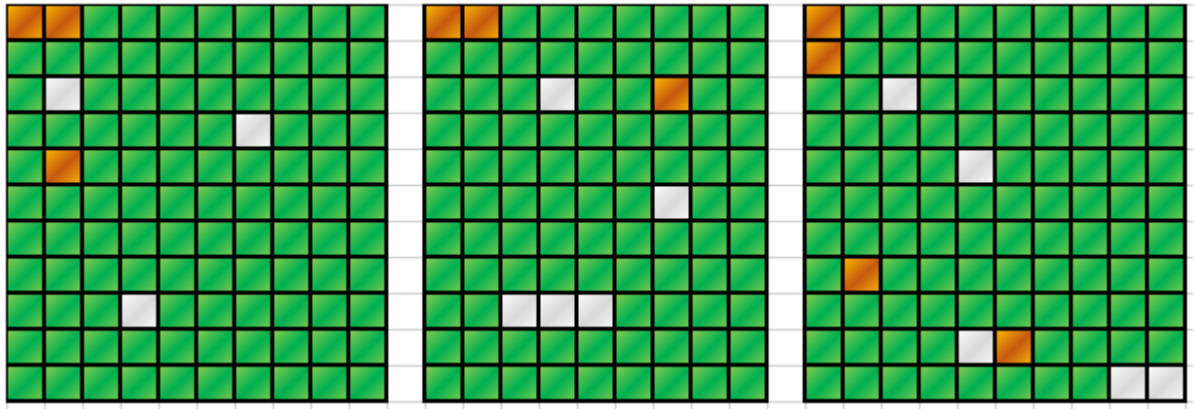
Smoke tests vs. Feature/Regression tests

- Smoke test suite:



- Purpose:
  - Run fast and check very basic function
  - Verify a feature isn‘t *completely* broken
  - To enable *frequent* checkpoints
  - To be fast, it *must* be minimalist
  - If minimalist, it‘s coverage *must* be small

- Regression test case suite:



- Purpose:
  - *Thoroughly* verify function of *entire* feature
  - Closest to 100% test coverage *as possible*
  - Function + integration + sanitization
  - Regression test case suite = Feature‘s Functional+System tests re-run on old code

BROADCOM

Lessons learned: The iceberg issues

- Don‘t think that Unit tests or Code coverage mean „tested enough“ – they‘re only the 1st half
- Do not mix Smoke and Regression tests: you will get a cat-dog which is neither fast (like Smoke tests) nor has the high coverage (like Regression tests)
- All new features must have Functional Tests Suite to verify all their aspects are good. This will sunsequently turn into Regression Test Suite for that feature
  - If you skip this, you will still have to invest the same effort into Regression tests later on, *but* will cause more defects to be introduced and reported by customers!
- Don‘t skip ahead and start with ad-hoc „testing“ without spending quality time on Test Design and Analysis

Dangerously misleading „testing lingo“

When you say „Testing“, you usually mean „Test Execution“

- The *visible* activity of:
  1. mechanically executing **pre-existing** test instructions
  2. observing & fetching results
  3. evaluating them vs. Test Oracle

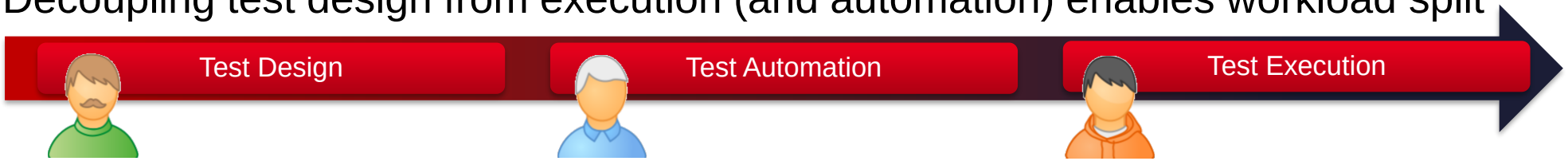
- Testing could be
  - Manual: Instructions-as-a-Doc for human
  - Automated: Instructions-as-a-Code for CPU
  - Where do the Instructions come from?!

- Crucial importance
  - Testing only as good as the Test Instructions
  - Anything mechanical could be automated

BROADCOM

„Test Design*“ is where these Test Instructions come from

- „Testing“ only as good as Test Instructions, which are only as good as Test Design
- Test Design effort thus determines:
  - Testing quality
  - Test coverage
  - Number and probability of defects caught
  - Real-world, customer-facing reliability of the product
- „Testing“ (test execution) is just a mechanical implementation of Test Design
- Why is this important?
  - Decoupling test design from execution (and automation) enables workload split



- *Industry correct term for test design: „Technical Test Analysis“*

BROADCOM

50% defects are data-driven, not code-driven

- Old IBM statistic
- Let‘s have simplest program.
  - Divide input 1 by input 2
  - 2 branches: if both inputs non-zero or else
- It‘s easy to achieve 100% „coverage“:

```
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
};
```

- Just these 2 test cases provide
  - 100% code coverage
  - 100% path coverage
  - 100% automation

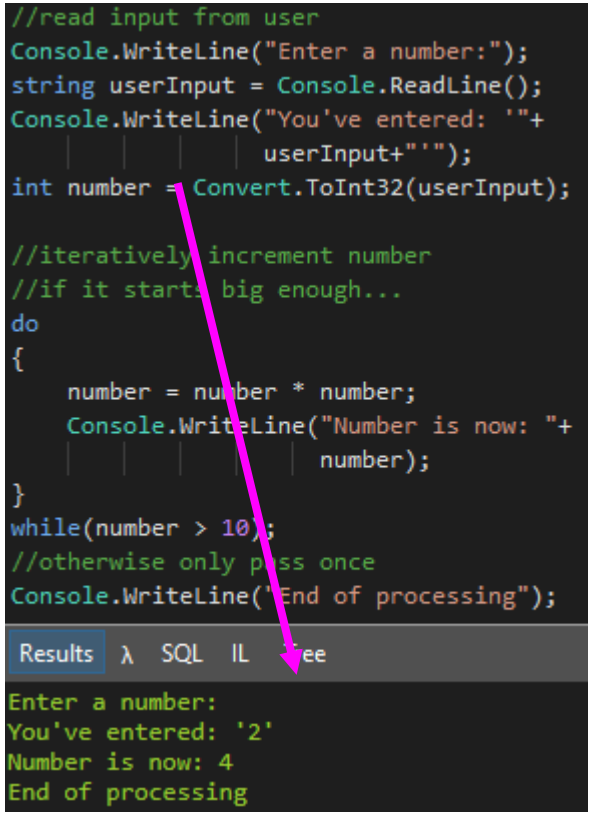
- Too good to be real? Because it isn‘t
- There are 2 classes of SW defects:
  1) Because what *was* in the code was wrong
  2) Because *something wasn‘t* in the code
- We need tests trying something *missing* from the code – handling of:

```
,{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, 0.33333333333333333333f} //periodic
,{4195835.0f, 3145727.0f,
    1.333820449136241002 5f} //Pentium FDIV bug
```

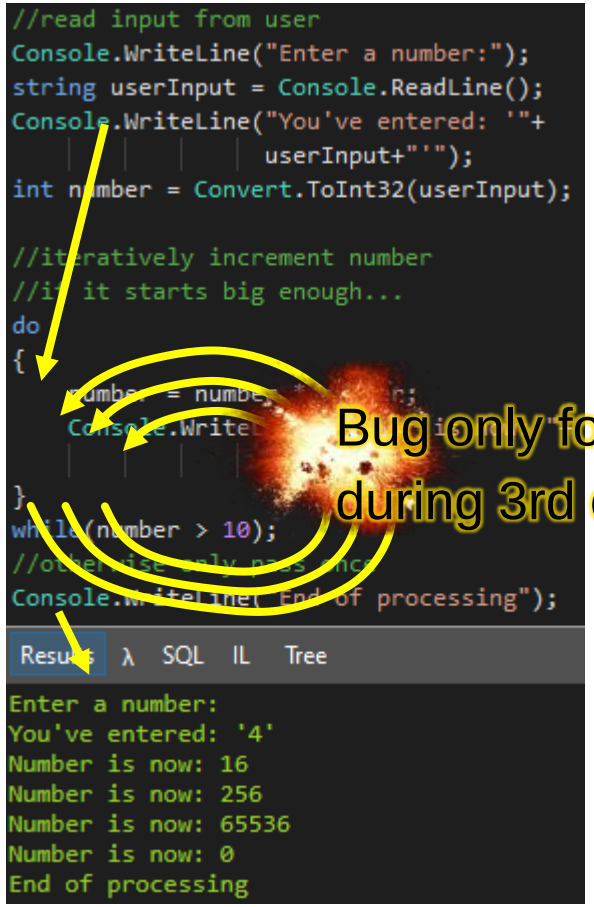

BROADCOM

Code Coverage vs. Path Coverage vs. Test Coverage

- Execute each statement

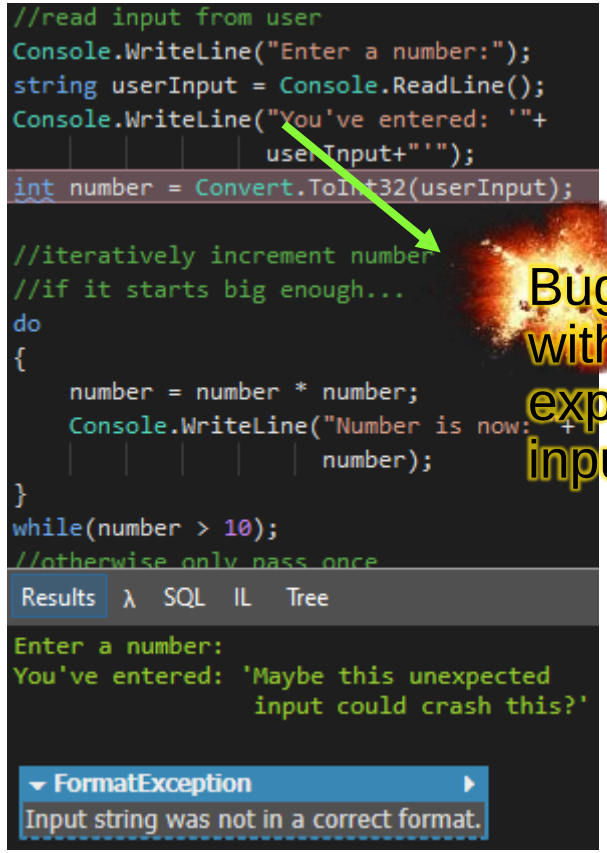


- Execute all flow paths



Bug only found during 3rd cycle

- Try every *possibility*



Bug found with un-expected input

Theoretical Assured Quality

BROADCOM

Testing metrics surprise!

- You‘ve just seen hands-on that:
  - Code coverage is weakest of metrics – cannot discover even defects in loops/cycles
  - Path coverage more capable, but still very narrow-focused
  - Because by design, nor CC nor PC can detect the 50% of data-driven defects
- Then why everyone loves Code?
  - Because it‘s easy to measure and report via automated tools
- Does that mean that Code Coverage is useless? No... *if* you know the limitations
  - CC is a bad overall target, but very good checkpoint ➔ no code function was missed
  - CC is a must, but not enough – data-driven testing needs to build on top of it

BROADCOM

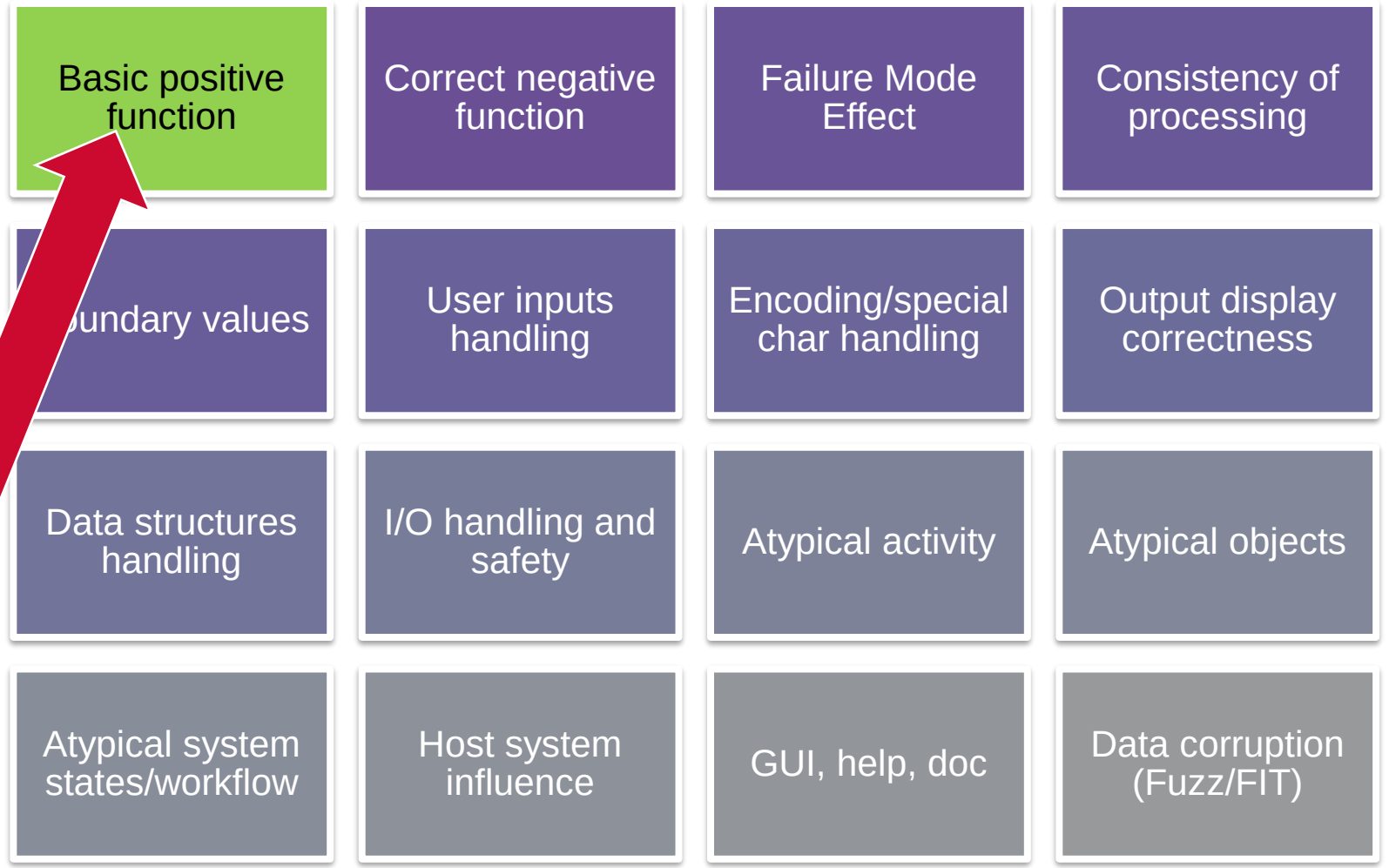
Even worse: 100% coverage is practically impossible

- 100% Code coverage cannot be achieved when code can‘t be reached by tests
  - Eg. Dead code, backend logic without API – or conditions impossible to artificially emulate
  - Code coverage > 90% *typically* is achievable (not always!), represents solid rock bottom
  - Code coverage *measurement* issue: few tools support Code Coverage of system/integ QA tests
- 100% Path coverage numerically and computationally impossible
  - Unlike linear Code coverage, Path coverage grows *exponentially* to conditions, loops, branches
  - Code with *just* 70 „IF“ statements = $2^{70} \cong$ 1,2 sextillion paths (~grains of sand on Earth)
  - Brute-force „all with all“ testing of code with 70 „IF“ statements = 10^{15} years (6000× age of Earth)
- 100% Test coverage – unreachable, untangible theoretical ideal
  - Testing *every possibility*? How do you enumerate, find, measure „all possibilities“?
  - Valuable insight: testing must go beyond just code, beyond just paths: user inputs, ENV conditions, host system influence, output correctness, displaying data, error messages...
  - ➔ instead of impossible Theoretical Test Coverage, we measure Practical Test Coverages as:
    - Percentage of Theoretical Test Coverage categories/concerns covered by Test Analysis
    - Percentage of Test Case Designs implemented into executable tests (eg. Automated)

BROADCOM

Practical Test coverage: The Basic 12 Aspects

- If „Test *every possibility*“ is impossible...
- You can get close by „Testing *every aspect*“
- How to do that?
  1. Cover all aspects
  2. Expand each to many test designs
  3. Implement designs to Test Cases
- „Basic testing“ only covers the 1st aspect!
  - Low test coverage regardless of other metrics



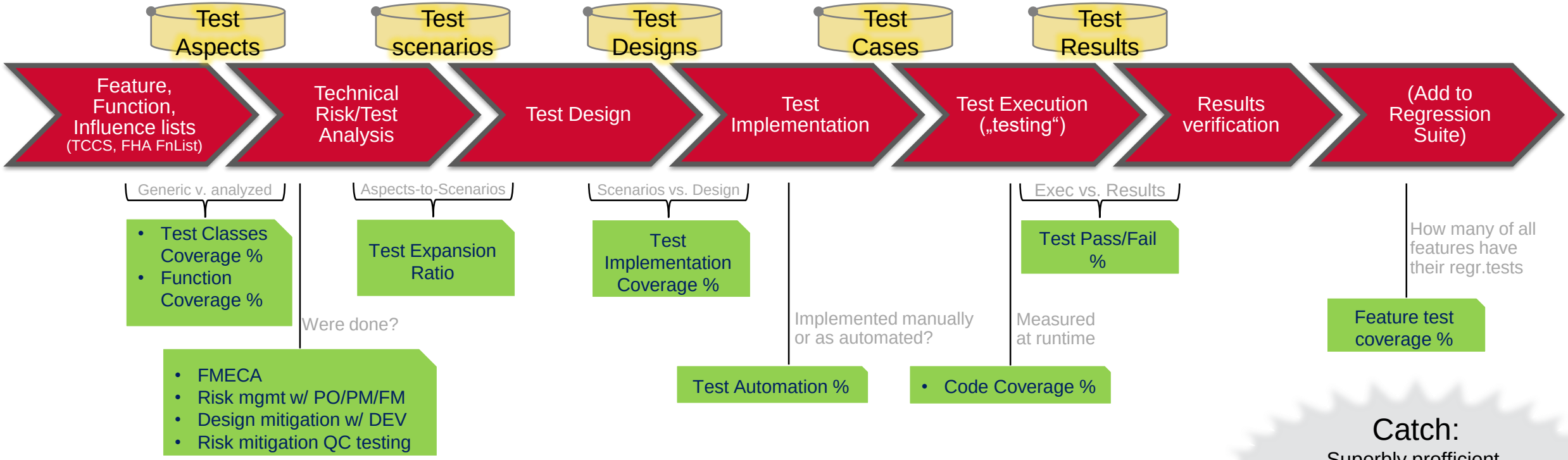
BROADCOM

Lessons learned: Misleading testing lingo

- Testing is only as good as Test Design/Analysis
  - Which is only as good as the test aspects it considers
- Critically important is testing of data aside of code – user inputs, data corruption...
  - Because even hypothetical 100% testing of code correctness covers just 50% of defects
- Code Coverage is not a magic bulet, nor the golden prize. It‘s auxiliary checkpoint
- The real „magic bullet“ would be Test Coverage... Which is however impossible
  - Also, cannot be measured in any practical way
- But you could get close to Test Coverage by testing all aspects
  - Of product design: using Function List
  - Of generic lists/lessons learned: Test Classes Cheat Sheet
- Test Design, Implementation and Execution/Testing can be logically separated
  - Eg. QA designs test scenarios, DEV scripts them to automated tests, QA runs+evaluates tests

Each Testing Workflow phase Has it‘s own metric

All the testing workflow, data and metrics explained



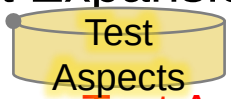
Catch:
Superbly profficient, experienced QC‘s can do all this as „background process“ in their heads...
To an extent

- Testing is only as good as the Test Cases it executes
- Test Cases are only as good as the Test Designs they implement
- Test Designs are only as thorough as the Test Scenarios they subject the product to
- Test Scenarios only cover the fractions of quality which the Test Aspects Coverage considers
- Test Aspects Coverage is only as complete as much the input Lists cover what could happen

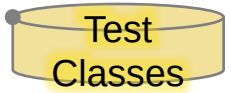
BROADCOM

Cheating the metrics: Goodhart's law and how to prevent it

- „When a measure becomes a target, it ceases to be a good measure“
  - People try to satisfy/improve the Target at the expense of other, yet equally important aspects
- Gaming the metrics (at the detriment of overall Test Coverage):
  - Increase Test Automation percentage:
    - Create less Test Designs ➔ lower the number of Test Cases to automate (--all other metrics)
  - Increase Test implementation Coverage:
    - Retain all 

, but automate just a fraction (--Test Automation)
    - Decrease the Expansion ratio ➔ create less Test Desigs to implement (--Expansion ratio)
  - Increase Test Expansion ratio:
    - Cover less 

➔ less --Test Scenarios to be expanded with more ++Test Designs each
    - Ignore some --Test Aspects as well as --Test Scenarios ➔ copypaste many minor variations of ++Test Designs (eg. Parameter values only) of the few remaining Test Scenarios
  - Increase Test Aspects coverage:
    - Don‘t create 

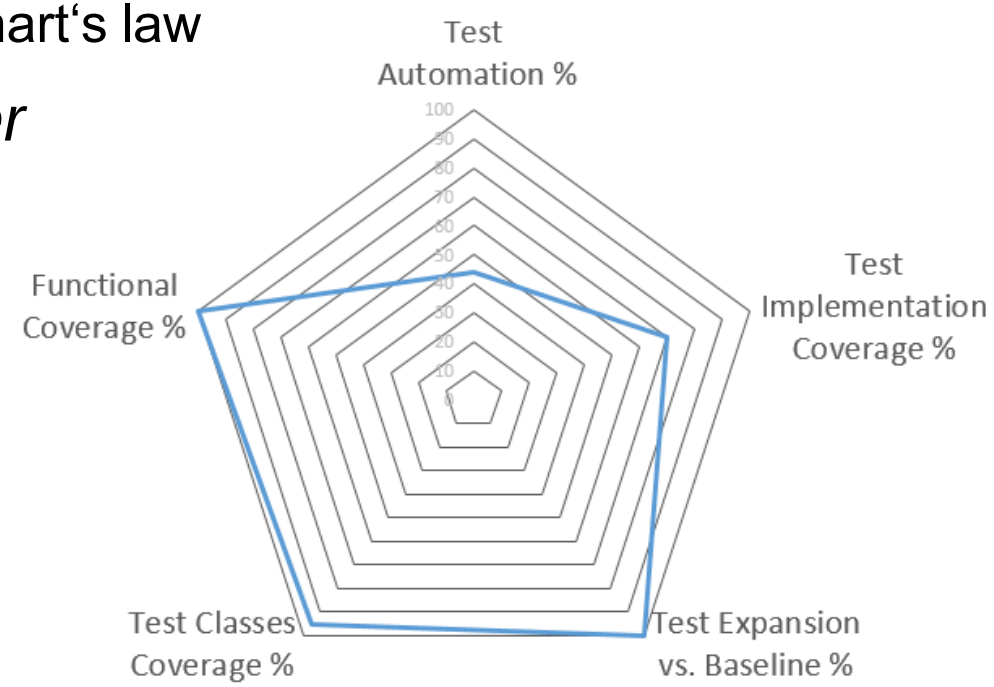
➔ some --Test Aspects „invisible“ without trace ➔ --Test Aspects to implement
    - Formally create a Function list, but skip/ignore some Pre-req or Support functions ➔ --Test Aspects
    - Don‘t use 

/ falsely flag some Test Classes as „N/A“ ➔ --Test Aspects to implement

BROADCOM

Only balanced metering doesn‘t incentivize bad quality

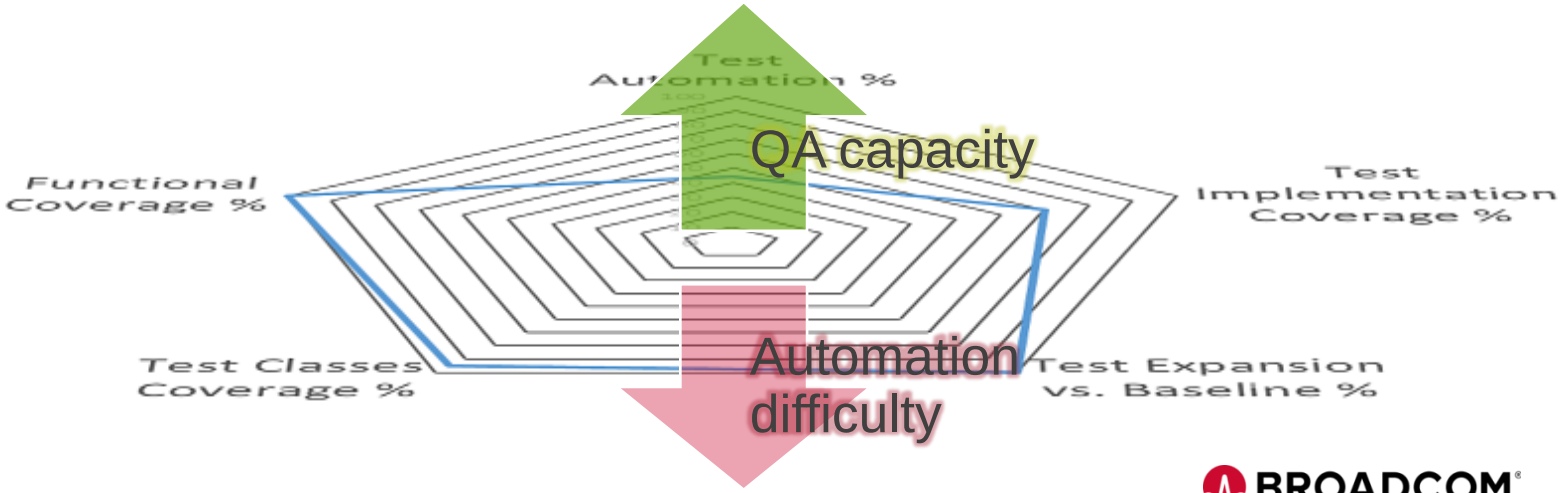
- Code Coverage is an objective measurement, but only ~5-40 % of Test Coverage!
  - 100% Functional coverage must trigger 100% Code coverage... But not the other way around!
- Test Coverage (=testing quality) is determind by 5 separate metrics...
- Of which each can be increased at the expense of others = quality!
  - If only one metric is selected, just that happens – Goodhart‘s law
- The test metrics only work *when used all together*
  - Instantly unmasking any „increase at the expense of decrease elsewhere“
  - Showing which aspect of quality is most lacking



BROADCOM

Sorry, there‘s even more to metrics

- The 5-axis Spider Graph just covers *technical* quality
- But then there are
  - **Team‘s QA capacity**
    (just 1× QA engineer cannot ever come close to 100% on all axis – not even 50%)
  - **Automation difficulty**
    (it‘s easy and quick to automate with APIs provided for all Test Design needs – product, host system and IO –, but hard and *very* slow without)
  - **Computational power**
    (if it takes 12 hours to run the automated tests, the debugging will take ages)
- So it‘s really 3-dimensional:



BROADCOM

Lessons learned: Testing metrics

- Each phase of Testing Workflow yields it‘s own metric
- Neglecting any phase of Testing Workflow decreases overall Test Coverage
- If you focus on any single metric, it will „improve“ single Test Workflow phase at the expense of others ➔ test coverage will decrease
- Testing metrics are ½ of equation – the other is QA capacity
- QA cannot focus just on testing and automation – *huge* overhead on maintenance of automation, infrastructure, orchestration...

Informal vs. Systematic quality

QA reviews: informal to systematic

- **Classical code review:**
  - Another DEV reads the code:

```
Console.WriteLine("Enter a number:");
string userInput = Console.ReadLine();
Console.WriteLine("You've entered: '"+
                  userInput+"'");
int number = Convert.ToInt32(userInput);
```

  - And provides feedback:
  - „Friend, this is dangerous code, because non-numeric inputs will crash it.
    Use this function instead:“

```
Int32.TryParse(userInput, out number);
```

- **Preliminary Hazard Analysis**
  - Origins in Aerospace
  - Assures top quality while lowering costs
  - Fancy name for:
    1. Identifying *all* feature‘s intended functions
    2. Identifying what could go wrong and how bad would it be (FMECA)
    3. Using this to improve test coverage, AND
    4. Suggesting feedback to Code/Architecture design
    5. So the feature is „armored“ against the bad problems
  - „Code review of design and risks – before coding begins“
    - Could be part of Architecture review, if Architects knew FMECA and Function Lists
    - But usually only senior QA/RAMS do this

BROADCOM

Input Lists, part 1: Test Classes

- Test Classes are:

hierarchical	• Logically organized with increasing precision
set	• A complete enumeration
of pre-defined	• Generic, pre-existing reference what exactly to test
test aspect considerations	• Solution to increase „Test all aspects“ test coverage

Test Coverage

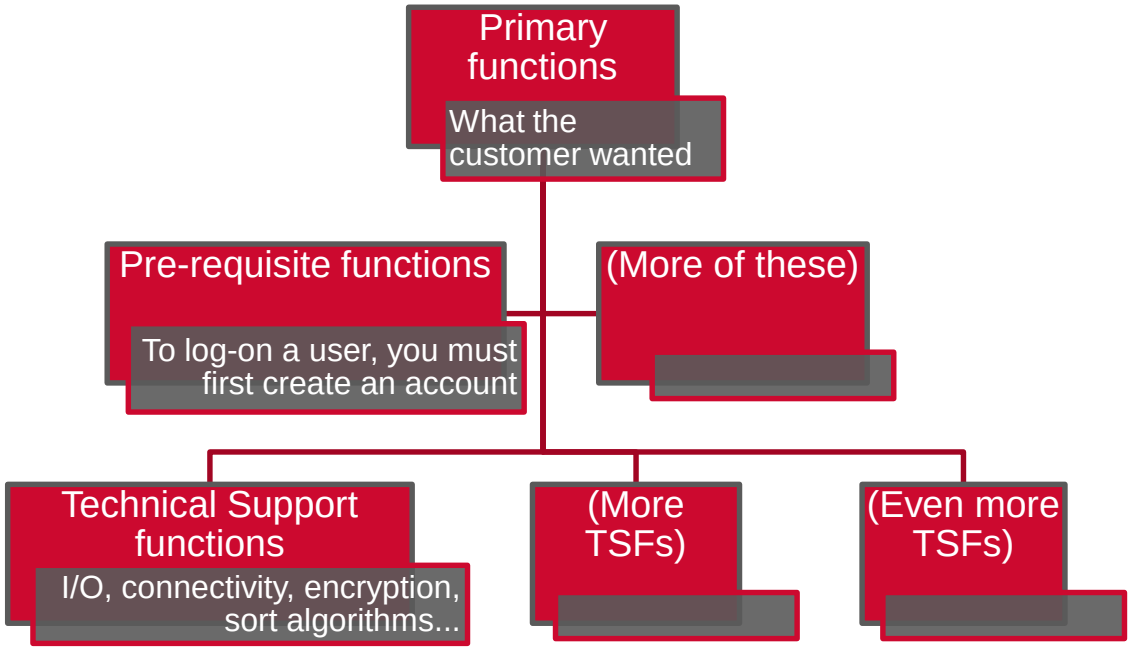
- They cover all 12 basic aspects to be tested.. And even more via sub-classes
- Generic list to start with
- Expanded to catch product-specifics

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 TCI 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**

BROADCOM

Input Lists, part 2: Function Lists

- Every software consists of:



- Customer/PO *should* know the 1st level
- Dev team must figure out the rest

- Once upon a time, all was pre-planned in:
  - Functional Specifications
  - Detailed Design Specifications
- Exhaustive doc ➔ didn‘t survive Agile
- ➔ Dark ages of ad-hoc improvization
  - „just start hacking code and add stuff later when you realize you need it“
  - ➔ lots of refactoring, glueing together, last-minute changes ➔ gaps, bugs, dead code
  - „just test somehow“ – not knowing what!
- Introducing Function lists:
  - DEV and QA architect and identify
  - *all levels of functions*
  - up-front but *lean* ➔ TDD, FnList as Test Aspects input list

BROADCOM

Simplified FMECA

FHA phase

- Method for discovering functional risks
- Allows risk-based testing without wild guesses
- Simplified FMEA cuts to the bones:
  1. For every function in Function list,
  2. Identify how it could fail
  3. Identify how severe would such fail be (using helpful SAE J-1739 10-point scale)

Fn	Failure mode	Failure effect	Sev
1.1	Failure to correctly separate parameter from value	Transferred dataset not encrypted with PE	8
	Failure to read and pass the entire value correctly		
	Failure to process the parameter (i.e. ignoring or rejecting it)		
	Failure to handle atypical yet allowed keylabel (e.g. “KEY.”)		
	Failure to capitalize lowercase keylabel		
1.3	Keylabel not shown although coded	Confusion and confidence loss	6
	Keylabel shown although not coded	False assurance – security issue	10

BROADCOM

QA as a rare resource

- Every hour of QA time translates to defect prevented – or escaped
- Any QA activity needs to be weighted based on Return of Investment:
  - How many defects would it prevent / allow to escape
- Most efficient: automated Feature Functional/Regression test case suite
  - Prevents dozens to hundreds of defects = customer cases when good Test Coverage
- Last efficient: single defect testing
  - Prevents re-occurence of a single defect
- Mid-way efficient: manual testing against Regression
  - To prevent so many defects, it would take months (took 3 months to 5 QAs in ACF2)
- Hence, top priority should be increasing *automated* Regression testing coverage
  - Because that tests the features, the product, *and* the defect fixes, freeing-up QA‘s hands
- But even bigger priority is covering new features with exhaustive Functional tests
  - Because not only it creates Regression test suite, but decreases defects Nr. in the new feature

BROADCOM

Test Automation caveats and planning

Avoid the „tip of the iceberg fallacy“

- Automation is Step №4, you need to start with Step №1

- Before you can start with automating test cases, you need:
  1. **QA infra** – the servers, libraries, deployment to run on
  2. **Test data source** – a SW repository of future automated test case‘s data
  3. **Test Framework** – tools, scripts and APIs to enable test automation
- Automation means
  1. **Fetch test data** – test definitions/inputs, standalone from automation Framework
  2. **Execute** – run the tested program with the test data as input
  3. **Fetch, parse** and evaluate Actual results, copare to Expected results
- When you automate, you must also
  1. **Debug** – test automation is coding/swdev of it‘s own, has it‘s own bugs
  2. **Orchestrate** – incorporate the new automated TC in Jenkins or other scheduler
  3. **Maintain** the automated tests over time (match product changes, Api changes...)

BROADCOM

Automation is a serious endeavor

- Selecting your Test Automation is akin selecting Corporate Enterprise Backend!
  - There will be 1000× more effort on the data in the system than on the system
  - ➔ The testing data will be 1000× more valuable than the testing framework
    (Without the testing data, you have no regression test coverage, would take years to rebuild)
  - ➔ Effectively „Vendor lock-in“: if you cannot easily port your Test Data
    to a different Test Framework... **You‘re stuck with it forever!** If the Framework goes EOS..☹
  - But what if the Test Framework doesn‘t support Automation of something your Test Case Design calls for?
    - A deadlock: you cannot change the Framework (as it would mean losing *all* the test data – Regressions)
    - But you cannot implement the test, if the Framework doesn‘t support what you need
    - Lose-lose scenario
- ➔ The Rules of Test Automation:
  1. Each test case automated = investment deepening Vendor-lock in the Framework it uses
  2. You shall not start Test Auto without serious planning and selection of the Test Framework...
  3. So that you‘re certain it will 1) be supported, 2) support *all your testing needs* for years to come!

BROADCOM

Test data ≠ Test automation

- As with testing: the „visible“ automation just implements something
- That something are Test Data definitions
  - the input values, flow of steps to be executed, expected results, host OS interaction...
- Test Automation doesn‘t start with the tool.
  It starts with Test Data Format
  - What are *all the things* your test cases want to do?

- When you *know* what your tests *will need* to do
  - Even in future – will new Product features require special testing capabilities?
- Then you can select THE automation framework *which supports* all the test needs
- And which will be licensed; supported; new team members can learn it; etc.
- Ideally separate test data from test automation tool (HP ALM warning example)

BROADCOM

Lessons learned: Test automation caveats

- Don‘t skip the foundations, preparations and planning - it *will* backfire
- Test Automation is investment & undertaking for many years to come
  - So respect the ramifications and plan accordingly
- Any test automation ➔ future maintenance overhead
  - Plan accordingly so that you can afford it!
- Test Framework matters. No size fits all. Select a test framework which:
  - Will provide API for any automation scenario the team/product will need
  - Will support future growth
  - Is resilient to tested product changes (tests not completely broken eg. by moving button)
  - Has assured future licensing or support (open source with huge community, core enterprise app)
  - Can be used by team members easily
  - Separates test data from test automation code, avoids vendor lock-in
- Before automation can start, you need QA infra (servers, VMs, datasources etc.)

BROADCOM