The Secret Life of Automation

Michael Bolton
michael@developsense.com
http://www.developsense.com
@michaelbolton

I'm Michael Bolton

Not the singer.

Not the guy in Office Space.

No relation.

I’m Michael Bolton

I help people solve testing problems they didn’t know they could solve.
And I help them learn how to do that for themselves.

This talk is based on personal experience and on interviews with friendly and generous colleagues.

Thank you to James Bach, Adam Goucher, Pete Houghton, Ben Simo, Andy Tinkham, and Anonymous

Questions? Write to michael@developsense.com

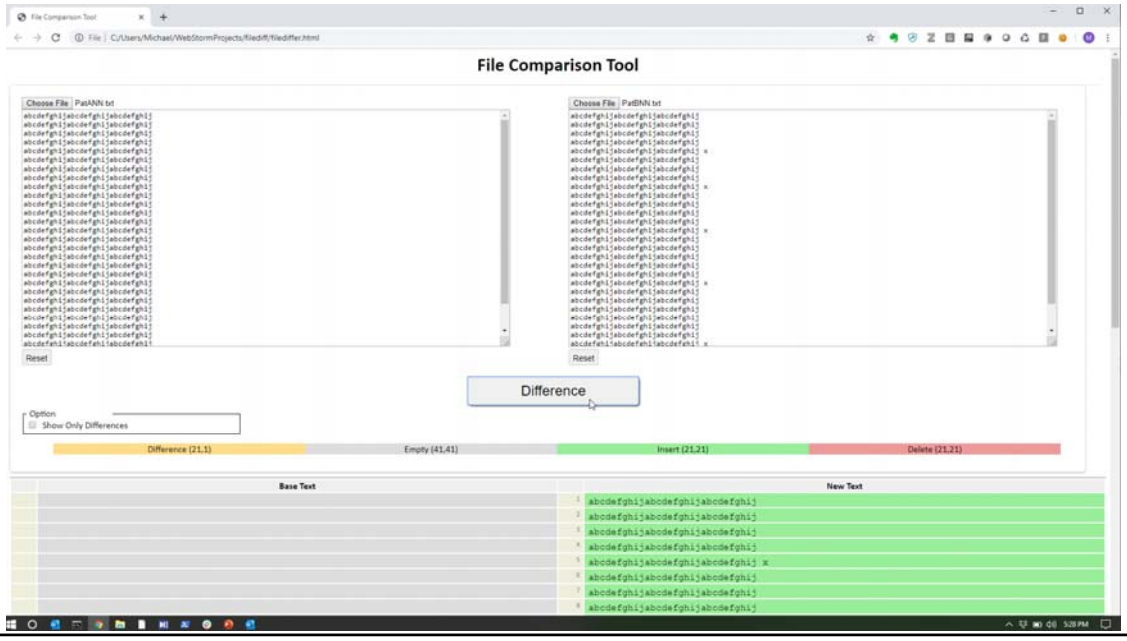
This talk is based on personal experience and on interviews with friendly and generous colleagues.

Selection Biases

People who don't need my help don't ask me for help.

People who *really* need help don't ask *anyone* for help.

Once Upon A Time…



Questions? Write to michael@developsense.com

In 2018, James Bach and I began to develop an exercise centred on FileDiffer, a single-page Web app that compare text files. We wanted such an app such that (among other things) people could manipulate and query its UI via tools (Ruby, Selenium, Excel).

I've worked professionally as a programmer in the past. But these days I don't consider myself a practicing programmer; I don't write test code every day. Without daily practice, I get rusty, and forget little details and nuances of the languages and the frameworks... even ones that I've written myself.

I wanted to produce examples of tools and automated checks that people might use to help test FileDiffer, using both good and not-so-good approaches. One example of a not-so-good approach is to rush into automating the product's behaviour through the GUI, before studying the product and learning about it.

So, I started with opening the browser and programming a couple of routines to find the input fields and the buttons, with the goal of producing totally simple checks and then extending them. Yay! — fast progress. I did one simple check of two chunks of text with NO differences, and another check for a really simple difference.

At several points I ran into testability-related bugs in the product. For instance, there are two Reset buttons in the UI, and both had the same element ID. I spent several minutes pondering and researching how to get around that problem (I wanted to be faithful to the product and not check the code).

I hacked away at studying aspects of XPath. Later I found out that James had fixed that bug in a new version of the product code. This happens in real projects! Often product code is being developed concurrently with the check code, and it's easy for the two paths of development to get out of sync.

It took me a couple of fiddly hours to get some check code running the way I wanted it to. No doubt if I were in better practice, that would have gone down to an hour or so.

Here's the bad part, though. At one point, the product was giving me output that I found surprising and confusing. Rather than investigating what was going on, I ignored my confusion. Instead, *I fixed the test code to match the product's output*.

I stepped in the Green Bar Trap!

When I didn't understand an aspect of the algorithm, I caught myself fixing my test code to match the product! In other words, I was building my own ignorance into the checks! This is a terrible idea.

Developing an understanding of a product can be *hard*, and the temptation to **JUST MAKE THE CHECK CODE RUN GREEEN** can be *overwhelming*, even for someone who famously warns people against exactly that.

This is important!

Testing is not only about functionality or technical correctness.

Those things are important to some degree.

But testing is really about discovering the fit (and misfit) between software systems and the people who use and develop them.

This is not about confirming that the product CAN work.

This is about *challenging* the product and finding problems.

SSSSSHHHH!

success = commercial advantage
failure = embarrassment
current state = **unknown**

What is not being talked about?

what people do in “test automation” work
what people do in *all* testing work
what people do in development work
what machinery really does
how tools could *extend* skills
concepts and forms

Questions? Write to michael@developsense.com

There's no sense in being precise when you don't even know what you're talking about.

John von Neumann

OK, so what *are* we talking about?

📖 testing

evaluating a product by learning about it
through exploration and experimentation and experience,
which includes to some degree:
questioning, study, modeling, observation and inference,
including…

📖 checking

the process of making evaluations by applying **algorithmic** decision rules to specific observations of a product

Call This "Checking", Not Testing

operating a product algorithmically to check specific facts about it…

Think "spelling checker"

means

Observe → **Evaluate** → **Report**

Interact with the product in specific, *algorithmic* ways to collect specific observations.

Apply *algorithmic* decision rules to those observations.

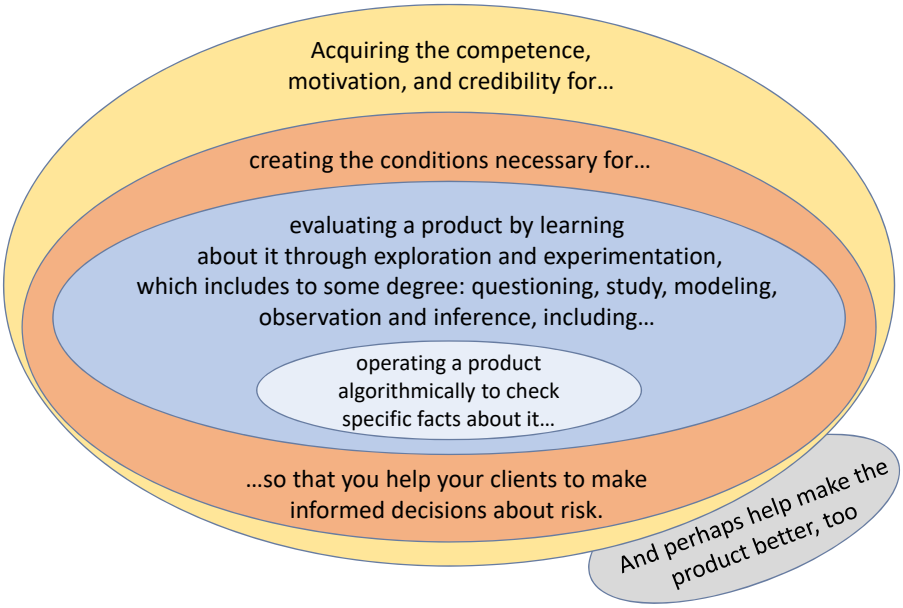
Report the outputs of the evaluations *algorithmically*.

Testing Is *More Than Checking*

- *Checking* is okay, but it is mostly focused on confirming what we know or hope to be true. And programming checks takes time.
- To understand our products and the risk of problems that matter to people, we must do more than output checking; we must *test*. And we must do that to design excellent checks, too.

See http://www.developsense.com/2009/08/testing-vs-checking.html

Testing is…



Why it's important to distinguish testing and checking

- Because *checking* is mechanistic. It can be made completely **explicit** and automated. It is *inside* testing. It is a *tactic* of testing.

People don't confuse biting with eating!

Biting can be done by tools… but eating can't.

Why it's important to distinguish testing and checking

- Because *checking* is mechanistic. It can be made completely **explicit**, encoded, and automated. It is *inside* testing. It is a *tactic* of testing.
- Because *testing* involves **tacit** and **social skills** that cannot be encoded. Testing skills and must be developed through socialization, practice, and increasingly challenging work, not via rote procedures.
- Because talk about efficiency and effectiveness makes for *very* different conversations when we're talking about explicit vs. tacit skills.
- Because for checking to be *truly* excellent, it must be embedded in excellent testing. Developing valuable checks requires skill!
- Programmers have resisted marginalization for years!
  (They no longer call compilers "autocoders" and programming languages are no longer called "autocodes".)

Healthy Perspective About Checks

We like checks. We use checks. They can help us affirm that the product can work, or if the product has suddenly stopped working in ways that are covered by the checks. Here are some caveats, though:

- Testing requires *learning*, and checks can't learn.
- Demonstrating that a product **can** work is far from showing that it **will** work under various kinds of challenges.
- The performance of a check can be automated. But designing, programming, interpreting, and improving checks—that is, the testing work that *surrounds* checking—*can't* be automated.

Testing Is *Social Science*

"Computers and their software are two things. As collections of interacting cogs they must be 'checked' to make sure there are no missing teeth and the wheels spin together nicely.

Machines are also 'social prostheses', fitting into social life where a human once fitted. It is a characteristic of medical prostheses, like replacement hearts, that they do not do exactly the same job as the thing they replace; the surrounding body compensates.

Harry Collins, Abstract, "Machines as Social Prostheses", EuroSTAR 2013

Testing Is *Social Science*

"Contemporary computers cannot do just the same thing as humans because they do not fit into society as humans do, so the surrounding society must compensate for the way the computer fails to reproduce what it replaces.

This means that a complex judgment is needed to test whether software fits well enough for the surrounding humans to happily 'repair' the differences between humans and machines. This is much more than a matter of deciding whether the cogs spin right."

Harry Collins, Abstract, "Machines as Social Prostheses", EuroSTAR 2013

OK, so what else are we talking about?

📖 automation n. “A high degree of mechanization in manufacture, the handling of material between processes being automatic and the whole system being automatically controlled.”

—Chambers Dictionary (iOS)

📖automation n. “the use or introduction of automatic equipment in a manufacturing or other process or facility.”

—Concise Oxford Dictionary

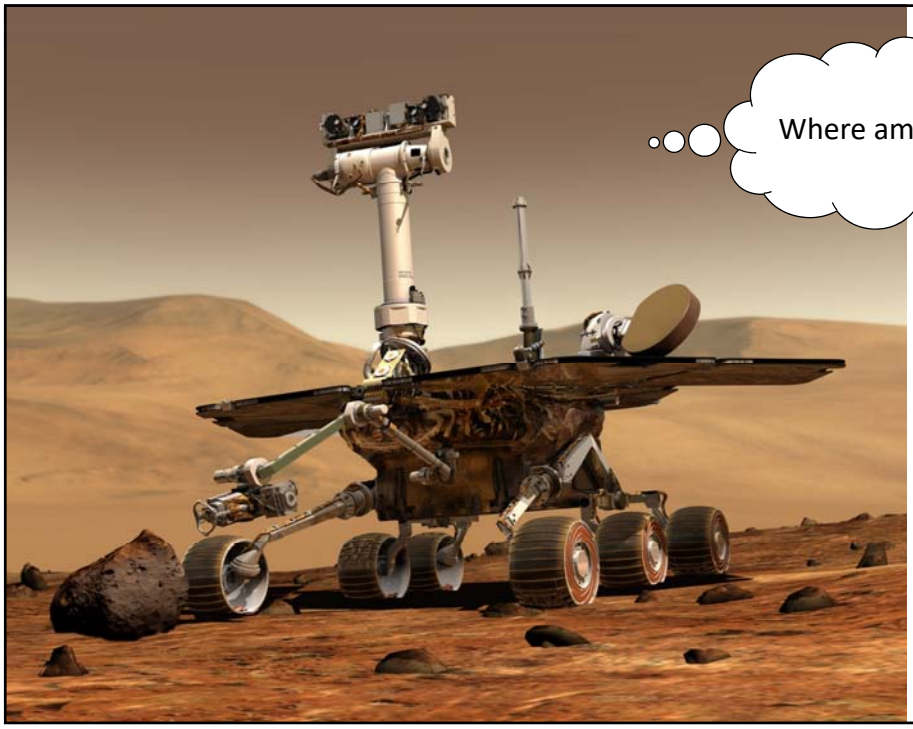
(How about “tool”?)

- 📖 a working instrument, esp. one used by hand
- 📖 the cutting part of a machine tool
- 📖 someone who is used as the mere instrument of another
- 📖 anything necessary to the pursuit of a particular activity
- 📖 a fool (slang)
- 📖 a despicable person (slang)
- 📖 a utility, feature or function available as part of *e.g.* a word processing package or database (computing)

—Chambers Dictionary (iOS)

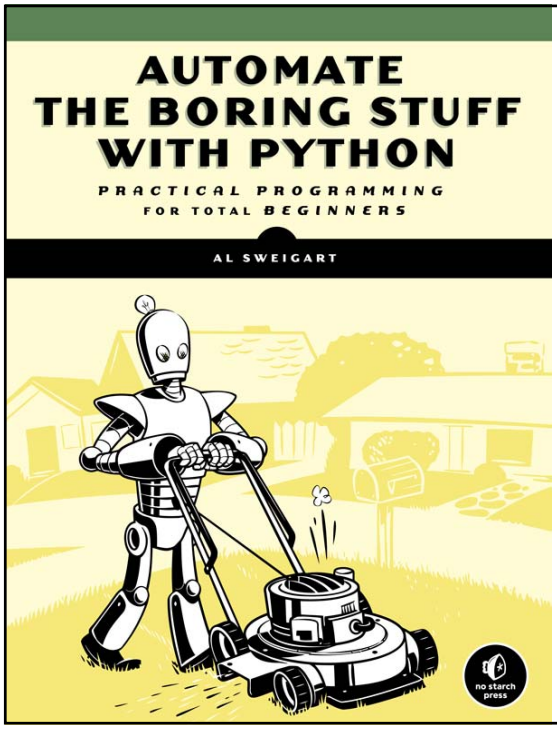
In Rapid Software Testing, we offer…

- 🕮 tool (n.)
  - 🕮 any contrivance used to fulfill a human purpose
- 🕮 test tool (idiom)
  - 🕮 any tool used in the service of testing
- 🕮 automation (n.)
  - 🕮 1. any process *entirely* performed by a tool
  - 🕮 2. a tool capable of such performance
- 🕮 exploratory testing (adj. + gerund)
  - 🕮 testing



The Mars explorers don't explore; *we* do, using these very sophisticated tools.

The Mars explorers don't know they're on Mars!



What can we say about this cartoon?

- The robot is almost certainly electrically powered, but the lawnmower is set up with a gas engine.
- The robot is humanoid, using a lawnmower's human interface. Why?
- If you wanted to automate the process of mowing the lawn, why not make things simpler and get rid of both the humanoid robot *and* the human user interface?
- If there are obstacles on the lawn, we can't see them.
- There's no bag to collect the cuttings, so depending on what you want, there may be some raking to do.
- There are no *people* in the picture. There are no sidewalks, either.

But let's acknowledge that the cartoon is *not to be taken seriously*. It's whimsical and silly. That's OK. The trouble is, we often seen automation in testing considered in this cartoonish way.

Photo source: https://www.amazon.com/Robomow-RS622-Battery-Operated-Mower/dp/B00IGXYBFI

This is a much more sensible design, but we're not out of the woods yet.

In order for the mower to understand where it's supposed to go, it needs a sensing wire to be put down around the edges of the lawn, and around any obstacles inside the area. The reviews are decidedly mixed; half are five stars, while one-quarter of them give only one star.

It's three times as expensive as a push mower (and this model is about half the price of models with a better rating). Putting down a boundary wire makes sense if your lawn and the obstacles in it don't change. If you have a huge space to cover, and you're not too fussy about how it gets covered, the robot mower might make a lot of sense. But the real point is that software isn't much like this anyway.

Here is one buyer's one-star review of the product.

My robot came with software version 0.86, it could not drive in straight lines, and it would randomly just freak out and say "not in working area", or "upside down" when neither was the case. By the way, the robot will think it has launched into the air and flipped over if it hits a tiny bump in my lawn. You literally need to have no bumps in your yard.

Eventually he changed his rating from one to three stars after the manufacturer shipped him a replacement for his first robot. He was better satisfied, but included these caveats: *This robot will not mow your whole lawn. Over the summer, I've had to slowly cede areas that the robot just can't handle. There were already a few areas like this, but I've had to add more and more area to the manual mow list. Sometimes something as simple as a tiny twig after fallen a rain will completely alter the robot's course over your lawn. It might cause the robot to get stuck where it never previously did get stuck.*

After 11 months, he provide one more update of the robot that he had come to call "Larry".
Larry started misbehaving - sometimes he'd drive in circles or just change direction at random and then start behaving normally. I didn't think much of it since I'd checked the boundary wire, the light on the base was green and he seemed to pull himself out of his funk after a minute or two.

UNTIL, it's everyone's nightmare - a few days later you come home and find a body floating in the pool. LARRY!!!

A Fable about the Roomba

ID 102881079 © Konstantinos A | Dreamstime.com

"We came home to find that the cats had crapped everywhere. While we were addressing what we discovered, I started up the Roomba. The Roomba smeared the cat shit all over the house.

The lesson: you have to clean up your shit before you automate."

Ben Simo

Secret:
A test cannot be automated.

Secret:
Automation IS NOT *testing.*

Secret:
Automated *testing* does not exist.
It *cannot* exist.

Secret:
Automated *checking* does exist, and it can be powerful. But automated checking cannot replace testing expertise and human interpretation, and must not displace it. Excellent automated checking REQUIRES testing expertise AND programming expertise.

Not Exactly a Secret:
If we (testers and others) keep talking about automated testing, people will continue to believe that it exists.

Also Not Exactly a Secret:
If we keep talking about automated testing, naïve people will continue to believe that testing can be automated.

Secret:
Whatever is being automated,
it's not the *testing*.
And it's not the user, either.

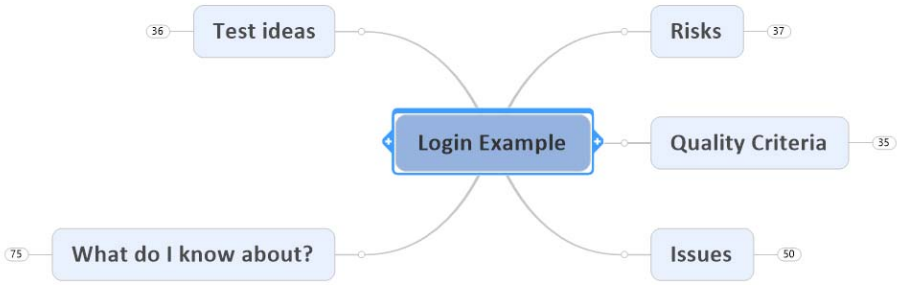
What I have learned from conferences, books, blogs, articles, and testing forums

The most thoroughly tested part of any application is

`You are now logged in.`

Questions? Write to michael@developsense.com

Login Stuff We MIGHT Want To Test



What I haven't learned about from conferences, books, blogs, articles, and testing forums

- risk (referring to the product)
- problem (referring to the product)
- bug, error, defect, etc. (referring to the product)
- quality
- value
- coverage
- oracles
- investigation
- discovery
- learning

What COULD be automated?

- MANY things performable by algorithms, including
  - setup and reconfiguration of test environments
  - provision of input
  - aggregation of input
  - searching, sorting, filtering of data
  - conversion of data from one form to another
  - altering sensory modes (visualization, sound)
  - comparable or parallel product oracles
  - probes to access to internal states of a program
  - randomization
  - mapping and perturbation of state machines
  - generation of alerts
  - pressing of buttons
  - checking output against specified results

What IS BEING automated?

- TWO things performable by algorithms.

  - pressing of buttons
  - checking output against specified results

Now, for the new word…

GEMPUB

Here it is in a bunch of different fonts…

GEMPUB

GEMPUB

GEMPUB

GEMPUB

GEMPUB

GEMPUB

What does this ugly, stupid word mean?

- **GEtting**
- **Machines to**
- **PUsh**
- **Buttons**

📖 **GEMPUB (n.) getting machines to push buttons**

Why the fixation on GEMPUB?

Popular test framing pattern #1

1. Testing is misconceived as being about *confirming that the product works*; checking.
2. There is a large test space to cover.
3. This space is currently being covered by (shallow) “manual test cases” (i.e., human checking).
4. These checks took a long time to develop and write up (hey, the binders alone were expensive).
5. THEREFORE we don’t want to throw out checks.
6. Since they’re already checks, we can reuse them (even though they’re not very valuable).

Why the fixation on GEMPUB?

Popular test framing pattern #2

1. Testing is misconceived as being about *confirming that the product works*; checking.
2. There is a large test space to cover.
3. People want *something* done to prevent total embarrassment (e.g. a missing screen)
4. Therefore, they'll often settle for showing that something exists and that it *can* work.
5. Getting a tool to *visit* every screen *sounds* cheap, and it's worth *something.*
6. As long as we can do that, deeper bugs are simply bad luck; too hard to find.

Secret:

People tend to focus on the How and the What of automated button-pushing, and on how it looks (it's dazzling!).
But they don't focus on the Why.

Secret:
You can use tools to map, probe, visualize, stir, disrupt, amplify, randomize… and EXPLORE.

Exploration and Analysis to the rescue!

By exploring and analyzing the product (with the help of powerful tools), we may discover

- real problems that matter to real people
- fast feedback to help address them
- specific new risks to focus on
- "broken leg" problems to influence strategy
- obstacles that we could surmount with help
- radical shortcuts for exploring specific risks
- ways to produce useful maps of the product
- obstacles that we could surmount with help

Testability to maximize the power of testing AND checking.

Time for some social science research!

- To what degree are the checks and their outcomes being analyzed?
- What is the nature of the bugs that are being found using automated checks?
- What kinds of things might be a really good idea to check?
- To what degree is the checking effort and its value being analyzed?
- How long does it take to develop, run, interpret, debug, and maintain ?
- Who is doing the automated checking? Testers? Programmers? Both?
- How much are testers, check developers, toolsmiths and developers collaborating?
- Do checks really "allow testers more time for exploratory testing"?

It's probably not possible to make very many useful generalizations, but it might be possible to learn some useful things from stories.

Stories can help to reveal the Secret Life.

Analysis at a client site…

- Client had 1100 automated checks, developed over several years
- These took approximately 24 hours(!) to run
- Of these 1100, 100 were regularly running red
- Of those 100, 25 were known environmental problems (therefore considered non-bugs)
- Of the remaining 75, about 10% (~7 total) were regularly false-positive reports (that is, non-bugs)

What questions would you ask?

Questions from My Analysis

? What are these checks actually checking?

? When you say “1100 checks”, is each check examining a single condition, or multiple conditions?

? Might it be more valuable to talk about coverage, rather than counting checks?

? Do the checks include variation to expose bugs?

? How often are the checks reviewed and analyzed for relevance and accuracy?

? Are the checks duplicating conditions already checked by programmers?

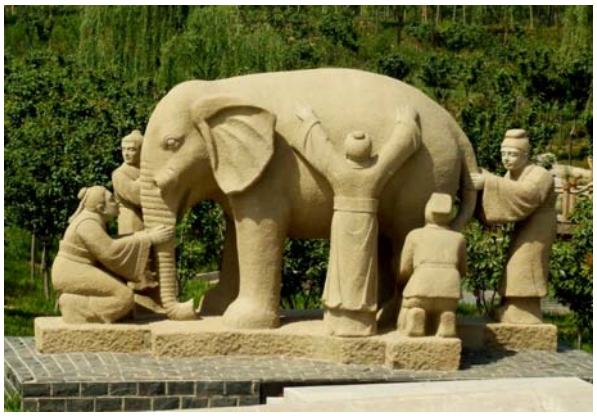
? Are the programmers doing *any* of their own checking?

? Why are the checks taking 24 hours?!

? What role does setup play in the timing?

Plus the invisible elephant…

- If 7% of the “failing” checks were unreliable, why presume that the “passing” checks were 100% reliable?



Secret:
Testing is confused with confirmation, and confirmation is a problem.

On Confirmation

Most of the technology of "confirmatory" non-qualitative research in both the social and natural sciences is aimed at preventing discovery. When confirmatory research goes smoothly, everything comes out precisely as expected. Received theory is supported by one more example of its usefulness, and requires no change. As in everyday social life, confirmation is exactly the absence of insight. In science, as in life, dramatic new discoveries must almost by definition be accidental ("serendipitous"). Indeed, they occur only in consequence of some mistake.

Kirk, Jerome, and Miller, Marc L., Reliability and Validity in Qualitative Research (Qualitative Research Methods). Sage Publications, Inc, Thousand Oaks, CA, 1985.

A Story of Confirmation

- Once upon a time, OrgB Technologies set up automated checks to confirm that a PDF was created. Checks showed that PDFs *were* being created.
- Somewhat later, OrgB discovered that although the PDFs were being created, they weren't being *launched* as they should be. Checks were then added to confirm that the PDFs were being started.
- Somewhat later, OrgB discovered that although the PDFs were being started, error reports came in the form of a PDF document. So checks were added to confirm that the PDF was the right size (the PDF spec was, at that time, proprietary and not available).
- Somewhat later, OrgB discovered that although the PDFs were about the right size, certain columns were not being included. So, after some time, OrgB found a library that was able to read elements of PDF files. This made it straightforward to determine that the right number of columns were in the file.
- Somewhat later, OrgB discovered that although the columns were being included, they were disappearing off the edge of the page.
- Somewhat later, OrgB discovered that the now visible columns contained invalid data…

Morals of these Stories

- Confirming the happy path often inadvertently focuses on *avoiding* finding problems…
- …but if you want to find the banana peels that the customers will trip over, you'd better
  - map the product (and iterate)
  - *vary* your paths
  - look for *problems,* in products, tools, and checks
  - learn from *every* problem

To what degree are checks and their outcomes being analyzed?

- In a very unscientific survey, my sources and my observation suggest “not very much” — especially not the ones that consistently produce green results.
  - Why? Perhaps because 160,000 “automated tests”, reviewed at a rate of 1000 per day, would take almost half a year to review completely.
- When they are being analyzed, I hear stories of
  - expensive and unhelpful formalization
  - busy-work
  - sunk cost bias

What is the nature of the bugs that are being found using automated checks?

- Since the checks and their outcomes are not being analyzed, the answer is unclear.
- From testers, several reports of bugs found during *development* of automated checks, but almost never thereafter.
- From developers using TDD, testing, checking, and development are all intertwingled, so the answer here is unclear too.

What kinds of things might be a really good idea to check?

- Units
  - Developer checks (TDD style) are relatively cheap to develop and maintain, since check development is done in parallel with programming and testing.
  - Feedback from lowest-level checks is very fast; never goes beyond the developer's machine.
- Data
  - Says one source: "I have a set of data that has to get massaged from its raw form to be usable by the software. There are 10,000 input files. They're processed, rendered into a certain form, put into a data structure, indexed, and stored. If one of those files is corrupted, it will give you an error message. There are 100,000 users, and they will have a high probability of finding the problem. With a single tester, you don't have that. Thus you may want a tool to visit all 10,000 of those."
- Sanity Checks
  - Checks for the existence of elements of the product, without a ton of intelligence about their content (that's hard)

What is the role of roles?

- Who is doing the automated checking? Testers? Programmers? Both?
  - Lots of variation here, but at the higher levels, it seems to be testers.
  - Lots of variation in programming skill
- How much are testers, check developers, toolsmiths and developers collaborating?
  - Lots of stories that add up to "not enough".
  - Lots of stories that suggest teams aren't taking advantage of developer expertise
  - Lots

Analysis?

- To what degree is the checking effort and its value being analyzed?
  - Not much, so far as I can tell from working around the world, and from colleagues.
  - This is not surprising; people are not doing this for the rest of testing either.
- How long does it take to develop, run, interpret, debug, and maintain checks?
  - See above.
  - For all practical purposes, nobody is talking candidly and publicly about this.
  - Failures are happening; they *must* be happening. But no one has an incentive to talk about it, and there are logs of disincentives.

Do checks really "allow testers more time for exploratory testing"?

- Due to all the other secrets, this appears to be a secret too.
- There is by nature *a lot* of overhead associated with automated checking. It's software development!
- There is little consensus that more ~~exploratory~~ testing is actually happening.

Ideas for Teams

- Development tasks *include* testing tasks.
  - The more you treat outsourcers as outsiders, the more likely their work will be misaligned with your work.
  - The more distance between “testers” and “automators”, the more likely that one won’t align with the other’s needs.
- Testing benefits from diversity and requisite variety. Diversify your team and your tactics.
- Learn from every bug.
- Revisit your checks; analyze their relevance.
- The belief that you MUST automate user-level checks suggests problems in your development process and your models of testing. Fix THOSE problems.

The Secret Life of Automation

Testing, by and large, exists in a constant state of existential crisis. At least at places where it is deemed a 'failure' or a 'cost'. So we choose to automate—but what, precisely are we automating? Is it only GEMPUB?

For Readers Only: Secret Secrets

Secret:
There are no flaky checks.
But there *are* flaky interpretations of results from checking.

Claim:
Tools can help us to do
more testing
faster
than we've ever done it before.

Secret:
Tools can help us to do
more lousy, shallow testing
faster and worse
than we've ever done it before.

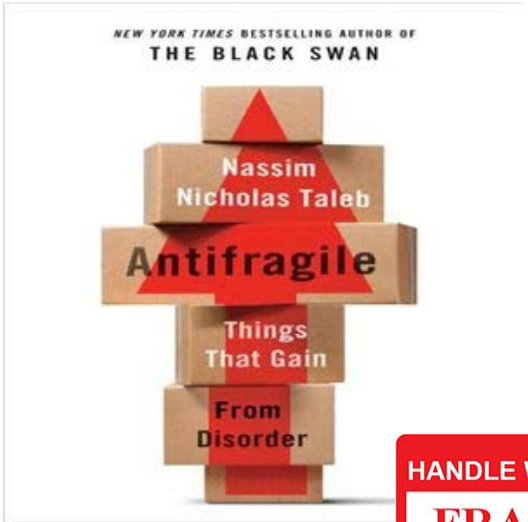
Secret:
When there are “flaky” checks, *something* becomes desensitized.

Secret:
When programmers, testers, “automators”, and toolsmiths are separated, secrets will multiply.

Secret:
People who have both the tester and builder mindsets are rare, and even they find switching mindsets to be hard.

Secret:
Many people try to make testing repetitious and over-focused, and therefore fragile.

Secret:
Successful testing should be an *antifragile* and *antifragilizing* activity.



Secret:
Don’t use tools simply to operate the product and demonstrate consistency. Use them to help probe, explore, map, perturb, visualize, reconfigure, tweak, generate data, parse, sort, search…

Secret:
We don’t know how to test until we’ve *tried* to test.
And we don’t know how to apply tools until we’ve *tried* to apply tools.

Secret:
Don't worry about building tools knowing you'll throw them away. Take advantage of the fact that as you're *building* tools, you're testing.

Claim:
Automated checking frees up more time for exploratory testing.

Secret:
Not necessarily. And certainly not when fixation on programming and maintaining automated checks dominates the testing effort.

Secret:
***Execution time* may be reduced, but there's also preparation, programming, debugging, troubleshooting, maintenance, analyzing failed checks, repair…**

Secret:
Preparation, programming, debugging, troubleshooting, maintenance, analyzing failed checks, repair, etc.
may afford learning… but will it be about things customers care about?

Technology Rule 1:

Any novel, non-trivial task will take longer than you think it will.

Technology Rule 2:

Rule 1 applies even after you've taken Rule 1 into account.

Technology Rule 0:
There will be bugs.

Secret:
Too often, tool work becomes solution-problemming.

Secret:
We shape our tools; thereafter they shape us.

—Marshall McLuhan

Secret:
You can't schedule epiphanies. But you can make room for epiphanies, and learn from every one. Learn from every bug.

Secret:
The Program Manager wants to know ONE THING above all: Are there problems that threaten the on-time, successful completion of the project?

Secret:
When you diversify your coverage and risk models (and talk about them), you'll get less pressure to attempt and rely upon confirmatory checking.

When you can describe all this and apply it skillfully like a professional, the amateurs don't hassle you.

Thinking about Better Test Strategy

Try replacing…	**With…**
Verify that…	Challenge the belief that…
Validate	Investigate
Confirm that…	Find problems with…
Show that it works	Discover where it *doesn't* work
Pass vs. fail…	Is there a problem here?
Executing test cases	Performing experiments
Counting test cases	Describing coverage
Automated testing	Programmed checking
Test automation	Using tools in powerful ways
Use cases	Use cases AND *mis*use cases AND *ab*use cases AND *obt*use cases…
KPIs and KLOCs	***Learning from every bug***

More Stuff: Use The Google

- My blogs
  - "On Green"
  - "On Red"
- With James Bach
  - "A Context-Driven Approach to Automation in Testing"