The Logic of Verification

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

James Bach
http://www.satisfice.com
james@satisfice.com
Twitter: @jamesmarcusbach

I'm Michael Bolton

Not the singer.

Not the guy in Office Space.

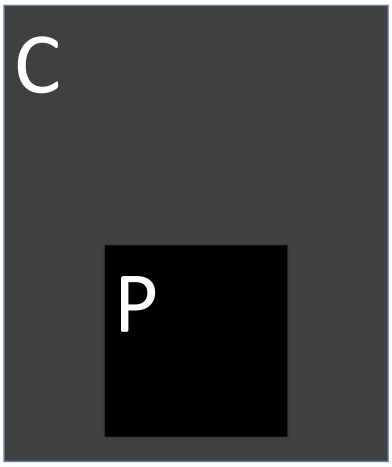
No relation.

?

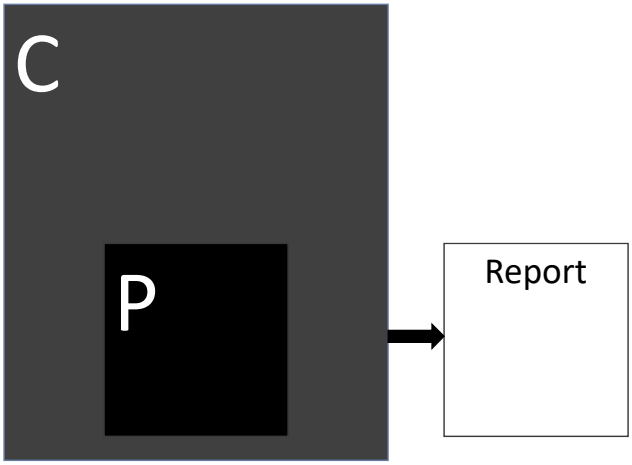
We have a product of uncertain quality.

P

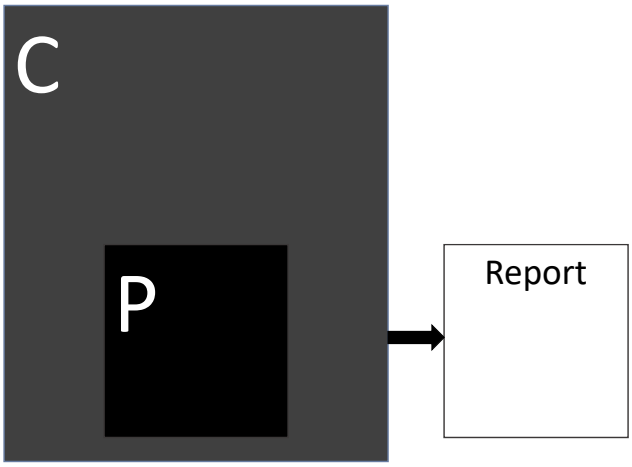
Let's call it Product P.



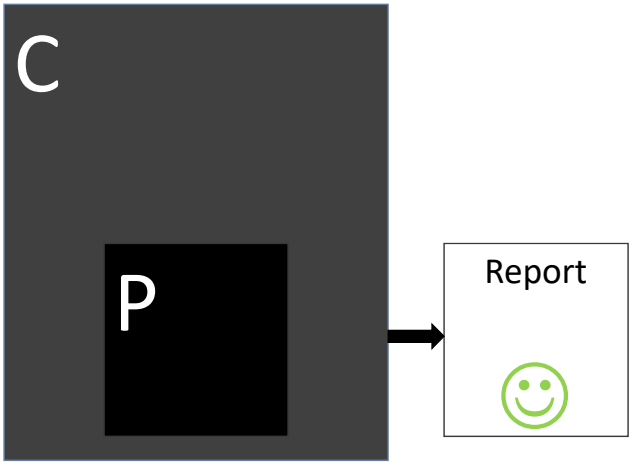
One way to evaluate P is to put it inside another system. We'll call that C (for "Checks").



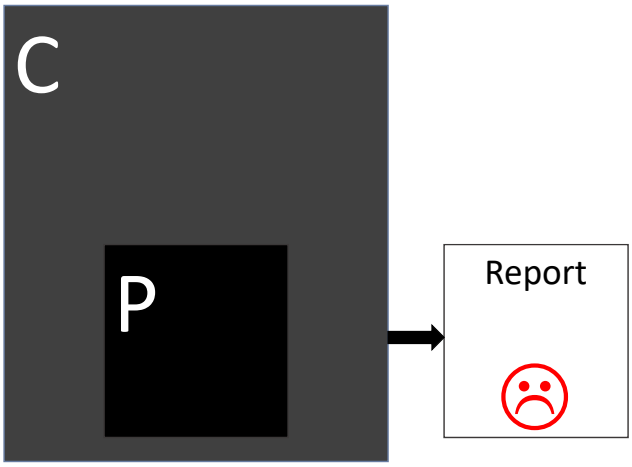
We design C to provide input to P; to operate it; to observe it; to compare the output to a specified result, and to report on the outcome of the comparison.



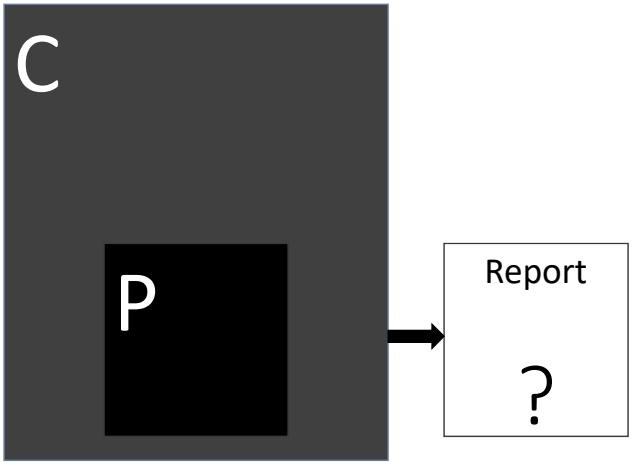
After this process, some people might be tempted to think this way:



"If C reports no problems,
we have verified that P is a good product."



"If C reports problems,
we have verified that P is a bad product."



But what is *really* going on here?

🕮 Verify (n.)

- To ascertain, confirm, check, or test the truth or accuracy of
- To assert or prove to be true
- To testify to the truth of, support (a statement, law)
- To check (items of data input) for accuracy eg by having the same data keyed twice, by separate operators, and then checked by computer for discrepancies (computing)

Weird!

—*Chambers Dictionary*

Verification (in the RST namespace)

🕮 Verification (n.)

1. The process of establishing the truth of a proposition
(this is universal, rather than specific to software)
2. In regulated software development, the process of comparing a product to its immediate specification

Verification is distinct from "validation"

🕮 Validation (n.)

the process of assessing a product against how well it fulfills its ultimate purposes

What IS Verification?

- Something exists.
- Some of what exists can be known.
- Some of *that* can be described in words.
- Some of *that* can be expressed as propositions which are either true or false.

Again: verification is the process of establishing the truth or falsehood of a proposition.

Verification isn't a feeling.

- Verification is reasoning via a logical process, within a logical system.
- X + Y = 2 has a **truth value** and **can be verified** as true or false **if** the values of X and Y are known, are numbers, and the conventions of arithmetic apply.
- X + Y = 2 may have a **truth value** that is **unverifiable** **if** the conventions of arithmetic apply, and X and Y are numbers, **but** those values are not known.
- X + ☯ = 2 **does not have a verifiable truth value**, if the conventions of mathematics apply.
- (We chose the ☯ symbol because it looks nice, and it starts with Y, but it stands for nothing in particular.)

DID work is not DOES work; CAN work is not WILL work.

In a system with a non-trivial state space, X + Y = 2 may be true ten times in a row, yet may be false on the next iteration. So…

- If you find X + Y = 2 is true even one time, then you have verified that it **CAN** be true.
- But unless you check **EVERY POSSIBLE** state of the system, *including possible states that you don't even know are possible*, you cannot verify that X + Y = 2 **WILL** be true.

Problems with Verification

- Propositions without a truth value can't be verified.
  - "Colorless green ideas sleep furiously." (huh?)
- Statements about the future cannot be verified until we reach that future.
  - "There are no bugs in the product." (*so far*)
  - "Our checks will find all the bugs in this product." (*we hope*)
  - "Customers will be satisfied with this product." (*we believe*)

Verifying Statements About The Future

- Go to a set point in the future
- Ask *all* customers "Were you satisfied with it?"
- Come back and report success! Hurrah!
- But even then, you can't verify that they would *remain* satisfied *after* you asked them.

Infinite Leap: situated fact ➔ abstract spec

"The product is not currently in a crashed state."

…is knowable here and now, but the fact that this is true does not mean that the product won't crash five minutes from now, or won't crash right now if I type the wrong key.

But what I care about is…

"The product shall not crash."

…is timeless and applicable to many situations. It cannot be verified empirically.

Infinite Leap: situated fact ➔ abstract spec

"I am able to read the buttons on this screen."

…is knowable here and now, but the fact that this is true does not mean that it will be true for all buttons, at all times, on all browsers, in every state, for every kind of person, under all lighting conditions.

But what I care about is…

"The product shall be reasonably easy to use."

…is timeless and applicable to many situations. It cannot be verified empirically.

Infinite Leap: situated fact ➔ abstract spec

"I recognize the signup screen and see nothing wrong."

…is knowable here and now, but the fact that this is true does not mean that it will be true for every situation where that screen should be displayed, that it is compatible with every browser, and that all the links and JavaScript do the right things.

But what I care about is…

"The correct signup screen shall be displayed."

…is timeless and applicable to many situations. It cannot be verified empirically.

Asymmetries: What We Can (and Can't) Verify

Verifiable	Not Verifiable
that there is a problem for some person	that there is no problem for that person
that we are not aware of a problem for some person	that there is no problem for any person
that the product did something under specific conditions that we have observed	that the product will do the same thing under conditions that we have not yet observed
that the product DID do something	that the product DOES do something
that the product CAN do something	that the product WILL do something
that we were aware of certain conditions we believed to be relevant to the test	that we were aware of all conditions relevant to the test
that a product does not meet a requirement	that a product does meet a requirement
that the product *appears* to meet a requirement to some degree	that the product definitely meets a requirement
that the product has not crashed	that the product will not crash
that we have not observed a problem in a feature so far	that there is no problem in a feature
that someone is currently satisfied with the product, based on what they know at the moment	that someone will continue to be satisfied when new knowledge is revealed
facts that might influence decisions about quality	the product's quality

Verification isn't exactly testing.

To say

"This product is very good"

is often like saying

"This product is very ☯ based on known variables X and Y, plus all our *assumptions* about *unknown* variables V_1 , V_2 , V_3 …V_{10000} … etc."

This is **unverifiable**, but it may be **testable**.

Testing is

way more

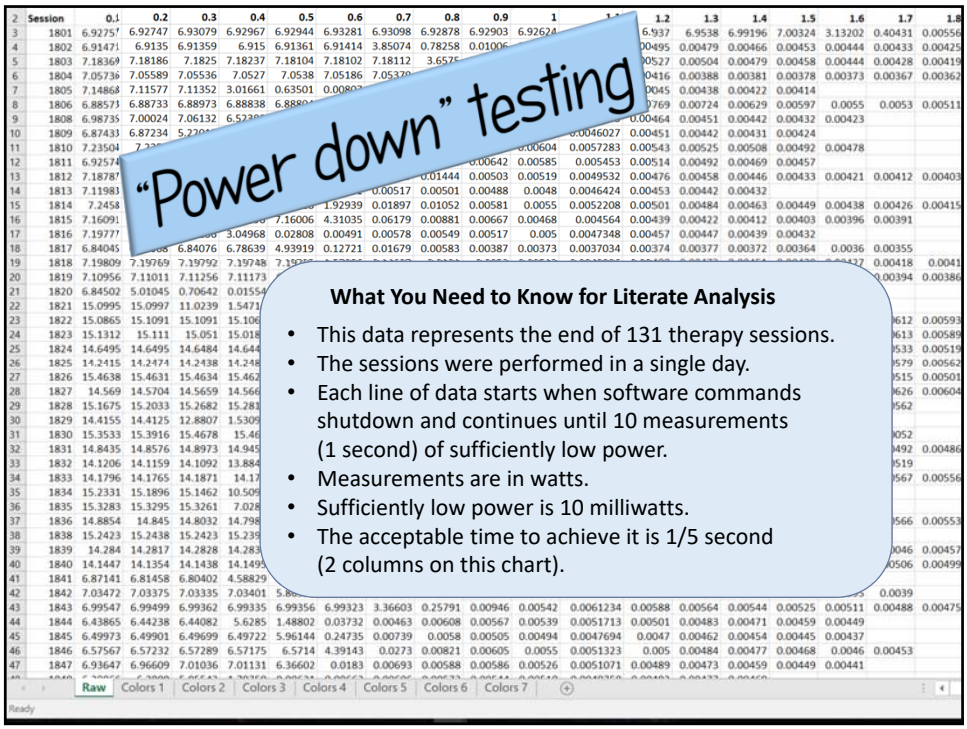
than verification

Oracle-Related Heuristics

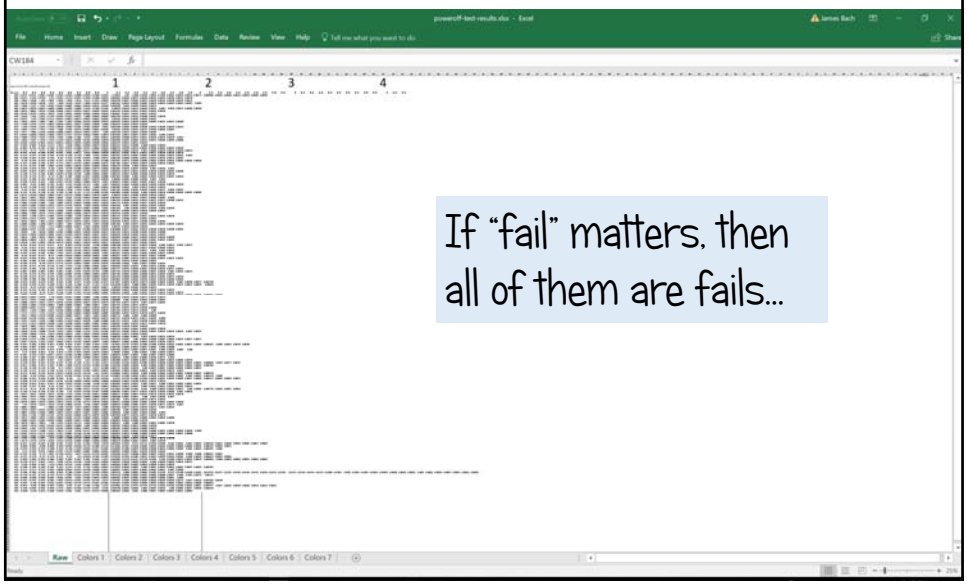
- An **oracle** is a heuristic for recognizing a bug when you encounter it.
- A **trigger heuristic** is a means of becoming aware that a situation requires your attention.
- A **radiator heuristic** is a means of conveying or representing information that you need to solve a problem.
- A **decider heuristic** is a means of deciding what to do to solve a problem.
- Thus there are **trigger oracles** and **radiator oracles** and **decider oracles.**

Verifications Can Be Good Triggers But Are Poor Deciders or Radiators

- A failing check definitely tells you that you have work to do. You must investigate. That's a **trigger**.
- Failing checks almost never **decide** that the software IS bad, because our first question is "Why did it fail? Could it be broken?" *The humans ultimately decide.*
- Log files, screens, and data displays are **radiators**. They are not subject to "pass/fail" but rather must be absorbed, interpreted, pondered, in loops.
- **Triggers** combined with **radiators** are an especially powerful combination.

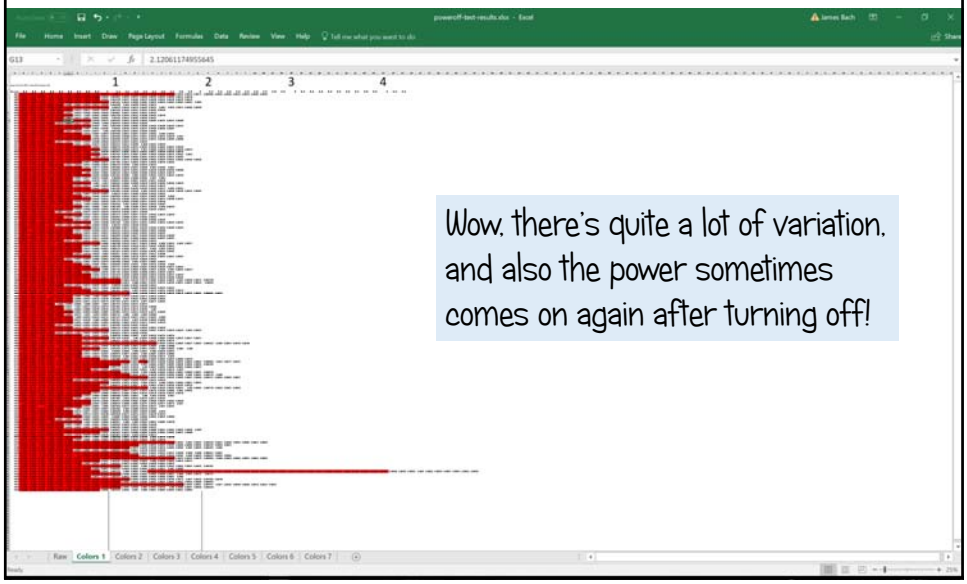


A “zoom blink” radiator oracle



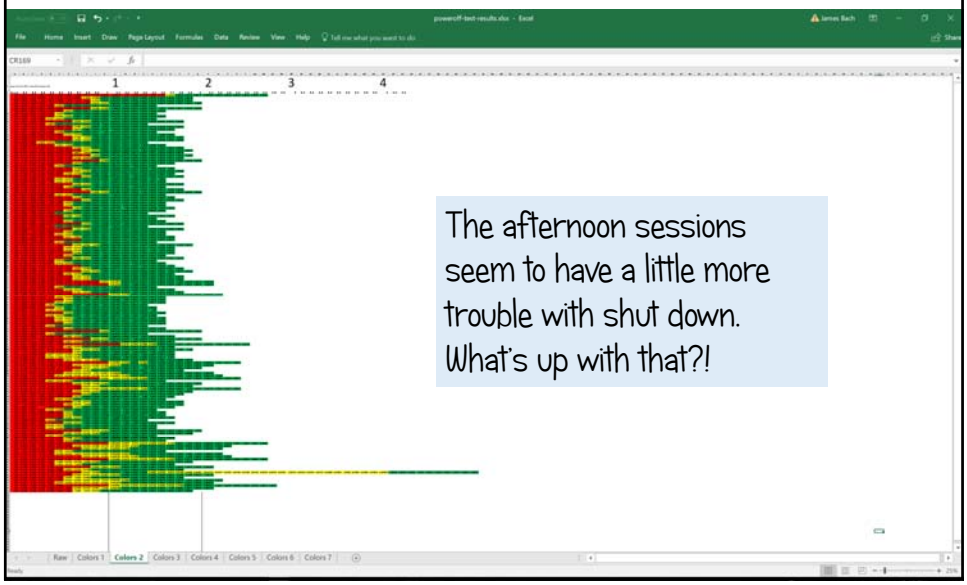
If “fail” matters, then all of them are fails...

Colored for >.01 vs. <=.01

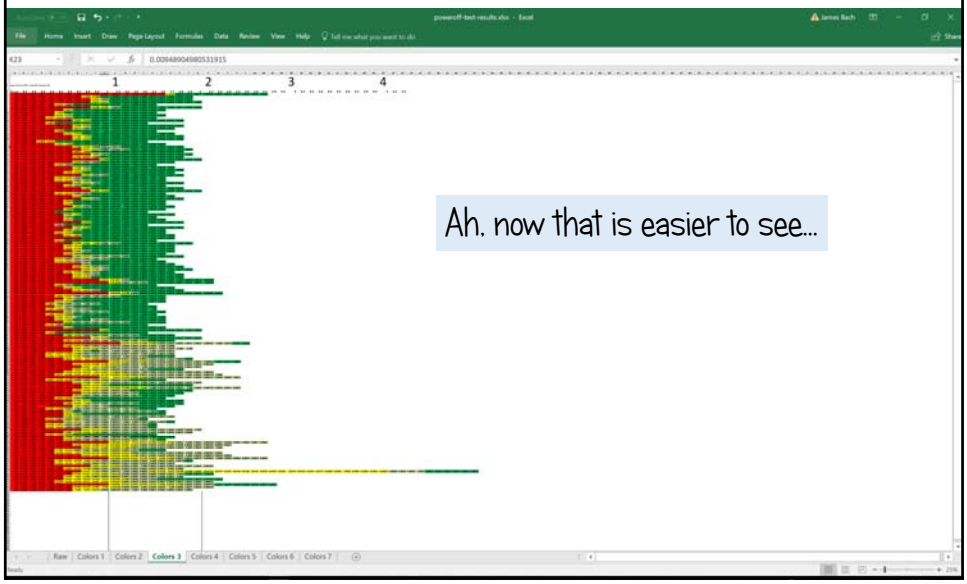


Wow, there’s quite a lot of variation, and also the power sometimes comes on again after turning off!

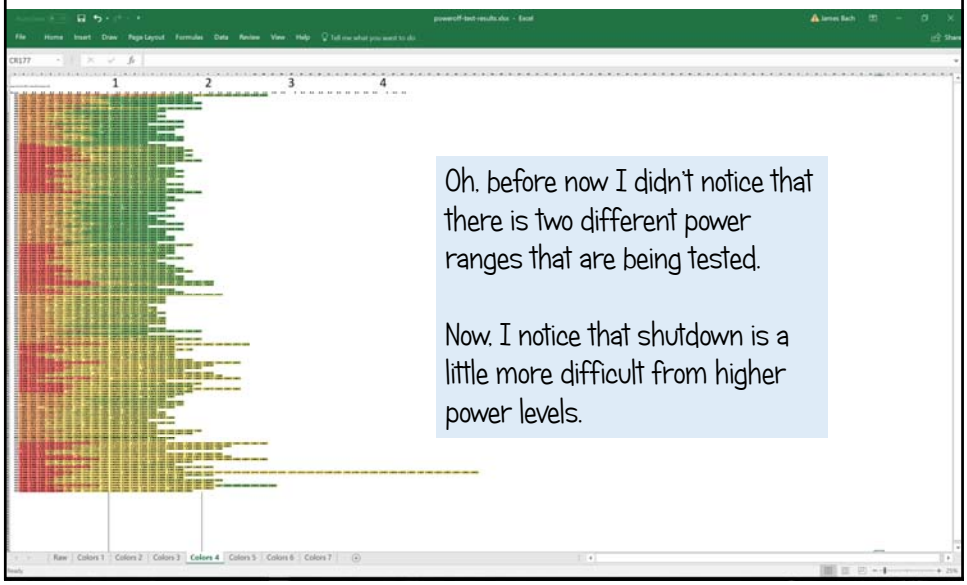
>=1 vs. <1 & >.01 vs. <= 0.01



>=1 vs. <1 & >.01 vs. =<.01 & >.08 vs. <= 0.08
"low bridge heuristic"



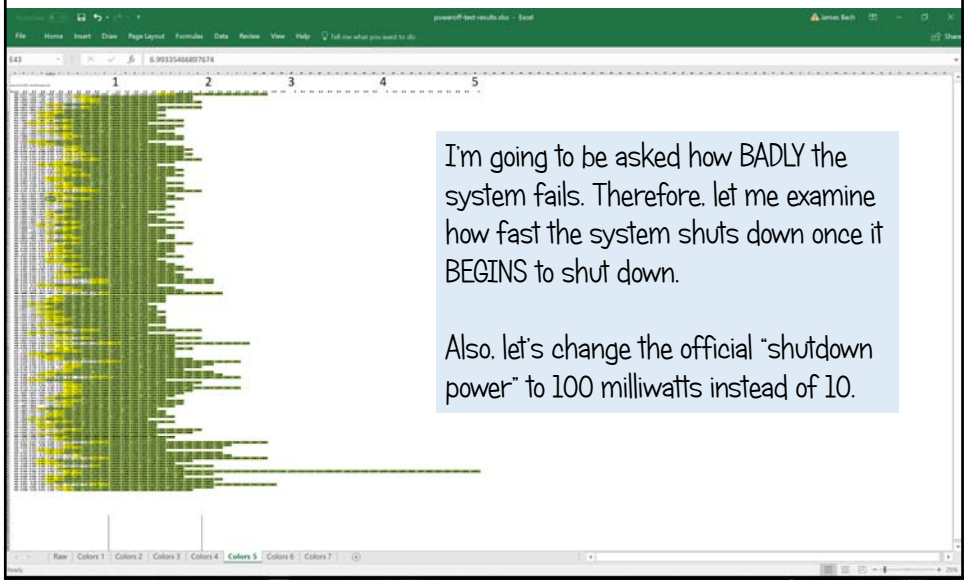
Defocusing: used default Excel coloring



Oh, before now I didn't notice that there is two different power ranges that are being tested.

Now, I notice that shutdown is a little more difficult from higher power levels.

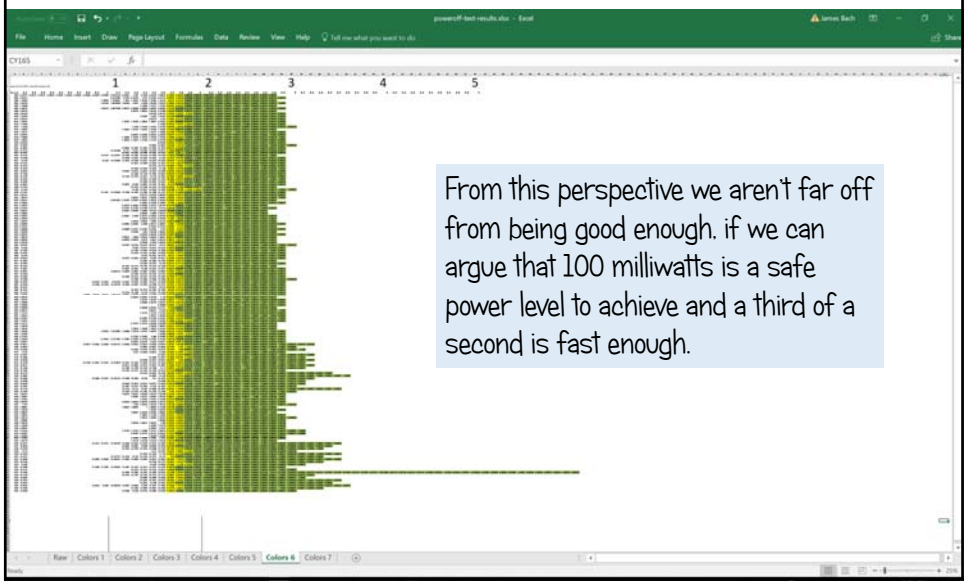
=< 90% of start vs. >= 0.1



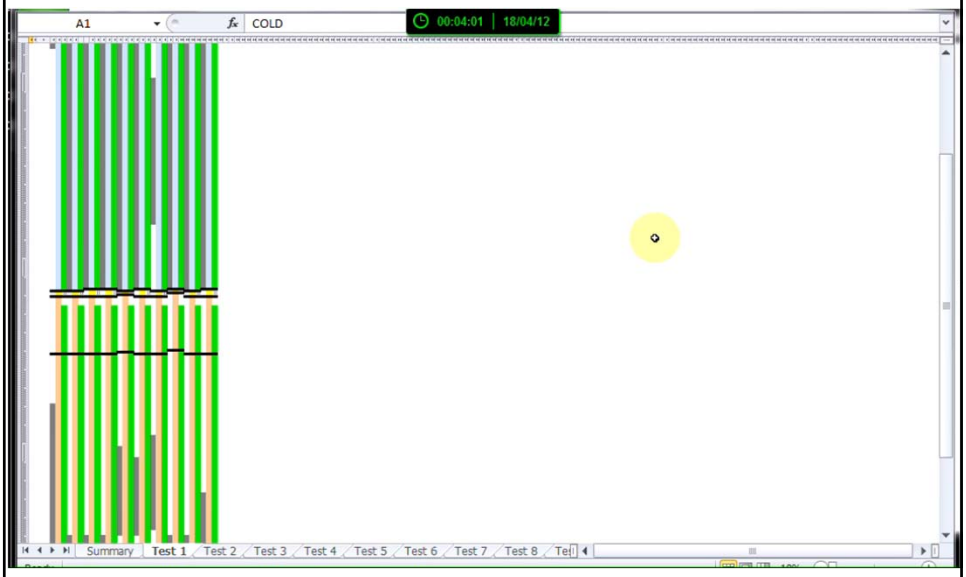
I'm going to be asked how BADLY the system fails. Therefore, let me examine how fast the system shuts down once it BEGINS to shut down.

Also, let's change the official "shutdown power" to 100 milliwatts instead of 10.

Centered on first =< 90% measurement



Triggers + Radiators Facilitate Testing

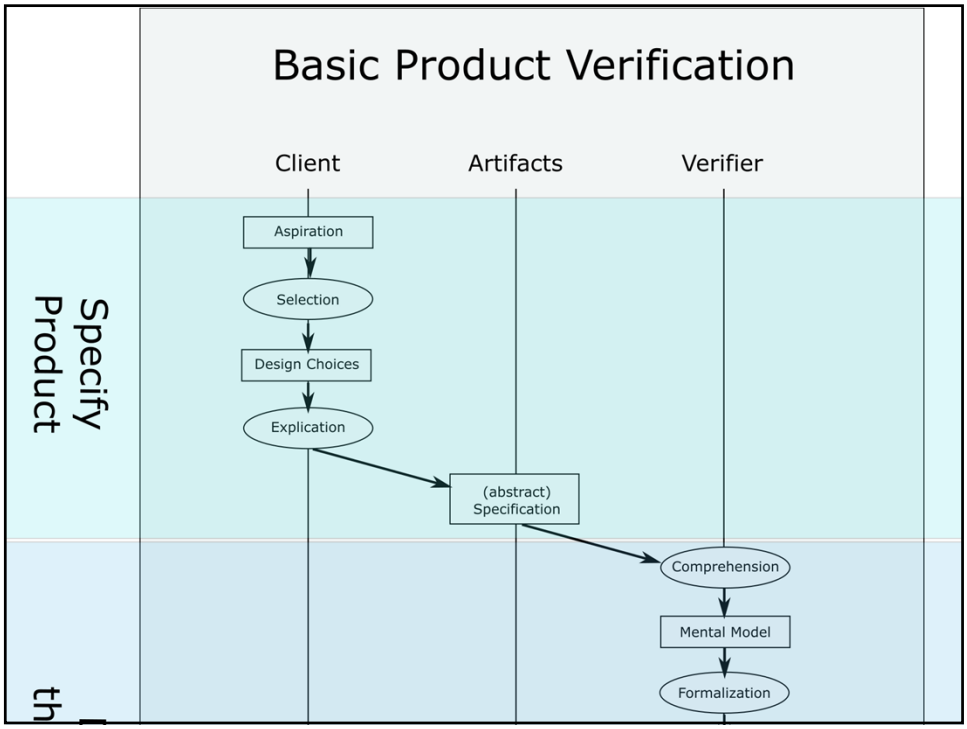


Triggers + Radiators Facilitate Testing

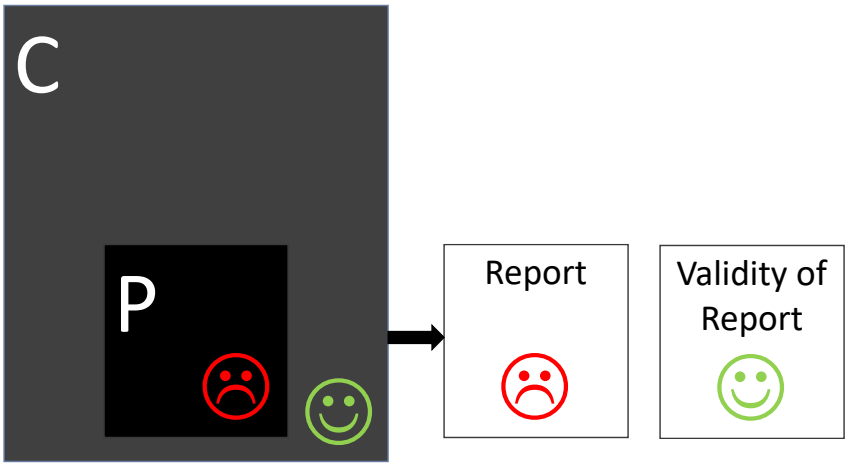
	A	B	C	D	E	F	G	H	I
89	STD. DEV.	n/a	0.41	1.83	0.29	0.47	0.00	0.00	18.08
90									
91	Test	Catheter	Marker Number	Catheter Type	Number of Electrodes				
92	6	6	6	Crescent	7				
93									
94		Time Until Within 5 Deg	Acc. After 3 Secs.	Time Until Restabilized	Stable Accuracy (HOT)	Stable Accuracy (COOL)	Stable Variance (HOT)	Stable Variance (COOL)	Sample Size (HOT)
95	1	00.6	00.78	08.7	-00.39	13.97	00.00	00.00	88
96	2	00.8	00.66	07.5	-00.42	13.97	00.00	00.01	103
97	3	00.5	00.88	09.9	-00.09	13.98	00.00	00.00	77
98	4	00.6	00.00	07.4	-00.91	13.10	00.00	00.00	101
99	5	00.7	-00.07	06.7	-00.69	13.90	00.00	00.00	110
100	6	00.6	-00.27	05.3	-00.71	13.91	00.00	00.00	123
101	7	00.6	-00.29	06.1	-00.81	14.30	00.00	00.00	114
102	8	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
103	9	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
104	10	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a
105	MEAN	0.63	0.24	7.37	-0.57	13.88	0.00	0.00	102.29
106	STD. DEV.	0.09	0.47	1.44	0.27	0.34	0.00	0.00	14.50
107									
108	Test	Catheter	Marker Number	Catheter Type	Number of Electrodes				
109	7	7	7	Circular	10				
110									
111		Time Until Within 5 Deg	Acc. After 3 Secs.	Time Until Restabilized	Stable Accuracy (HOT)	Stable Accuracy (COOL)	Stable Variance (HOT)	Stable Variance (COOL)	Sample Size (HOT)
112	1	n/a	19.06	12.1	16.90	14.65	00.00	00.00	52
113	2	n/a	18.48	11.0	16.50	14.66	00.00	00.00	61
114	3	n/a	18.86	11.8	16.90	15.17	00.00	00.00	53
115	4	n/a	17.54	08.9	15.82	13.90	00.01	00.02	82
116	5	n/a	18.28	11.2	16.40	14.65	00.00	00.00	60
117	6	n/a	18.30	09.7	16.10	14.45	00.00	00.00	74
118	7	n/a	18.50	11.8	16.30	14.27	00.00	00.00	54
119	8	n/a	19.16	12.7	16.36	14.43	00.00	00.01	45
120	9	n/a	18.98	14.4	16.84	15.97	00.35	00.00	33

Summary | Alerts | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Test 6 | Test 7 | Test 8 | Test 9

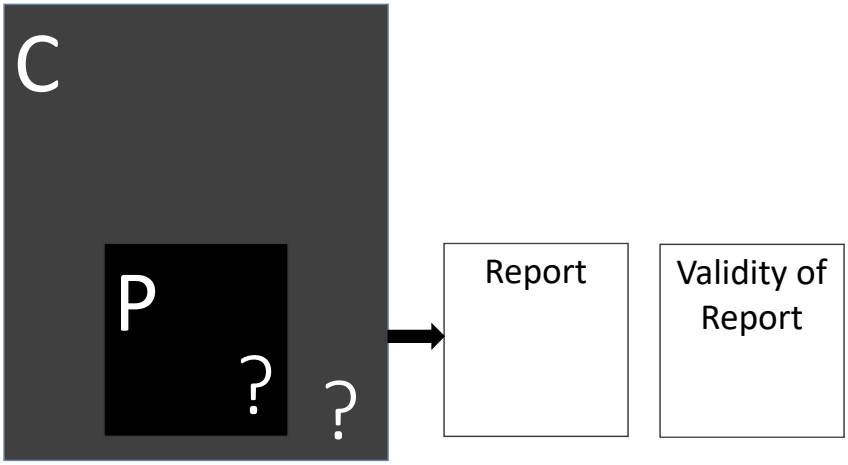
Entity	Definition
Client	Person who matters; whom we serve.
Aspirations	Ideas and values within our clients which relate to want they want.
Design Choices	Choices made by the client, or on the client's behalf, about what to ask for in the product, based on (possibly contradictory) aspirations.
Specification	Abstract statements that represent design choices
Assertion/Example	Situated proposition or artifact that is consistent with some specification.
Product	An artifact intended to fulfill the specification to a reasonable and acceptable degree.
Check (verification)	An algorithmic process that corroborates or refutes an assertion about a particular product in a particular situation.



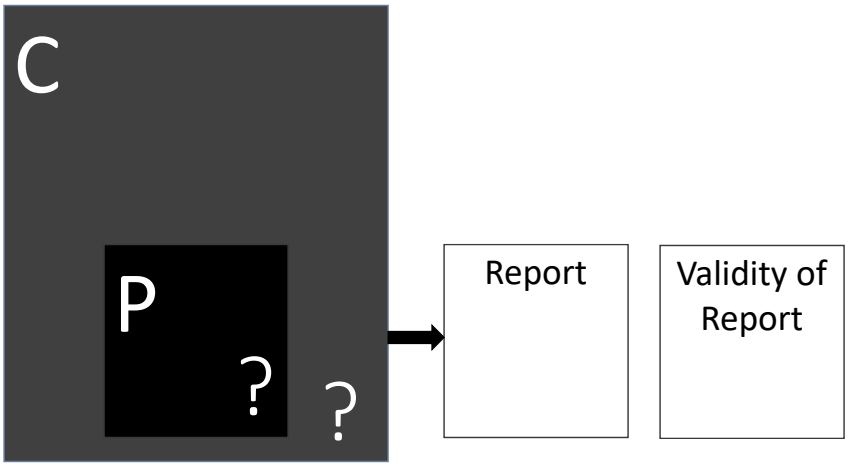
C reports that P is good. If and *only if* C is good AND P is good too, that report is valid



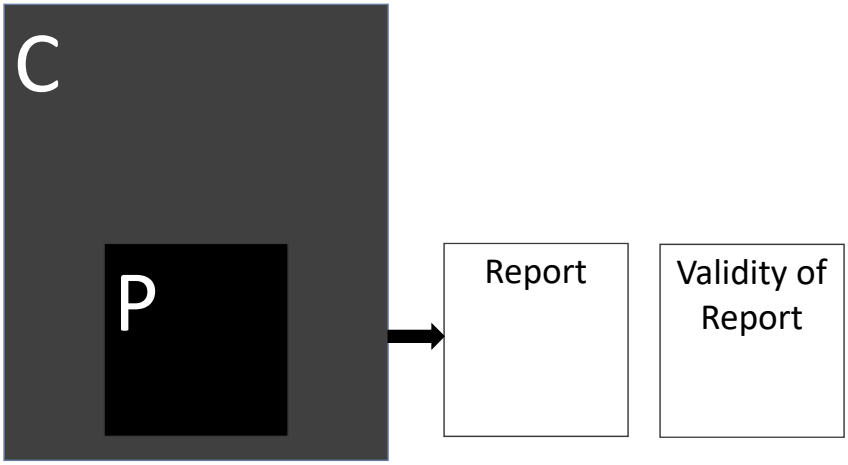
If C is good, it will correctly report that P is bad when P's quality is bad; C's report will be valid.



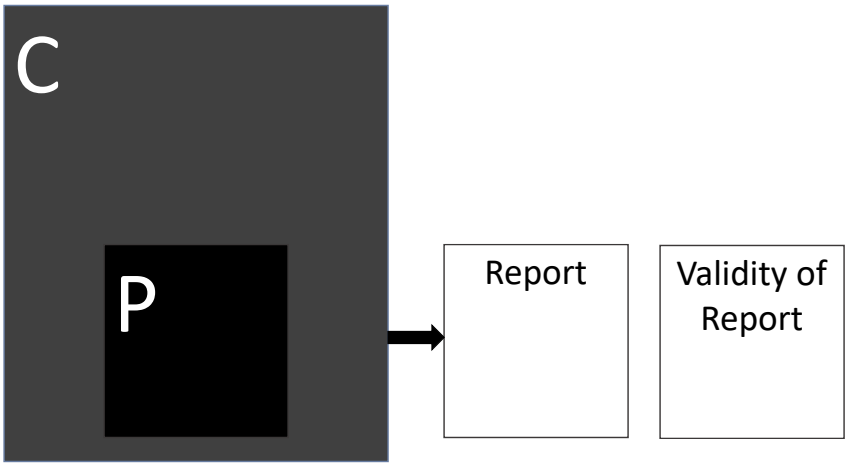
Code is often written quickly, or under pressure, or both—whether it's product code or check code.



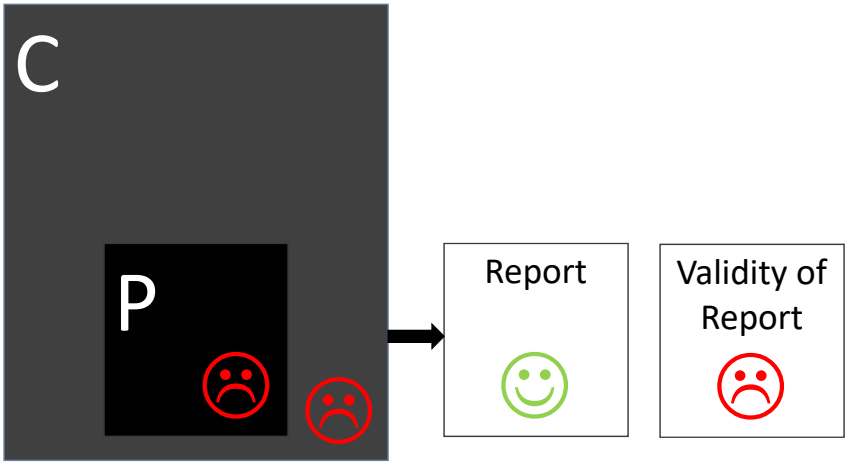
It is tempting to believe that
product code is more important than check code.
Customers don't see check code.



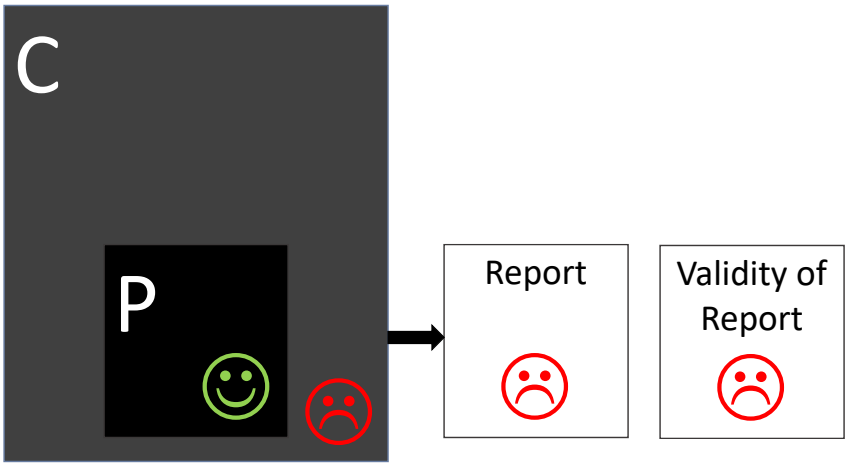
But conclusions we might make about P are risky,
because the quality of *both* P *and* C is uncertain.



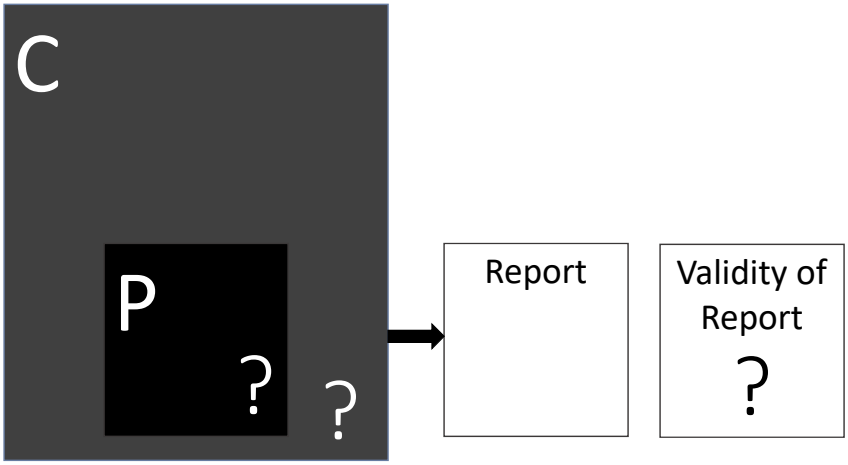
Conclusions we might make about P are even more risky when C is not developed carefully and skillfully.



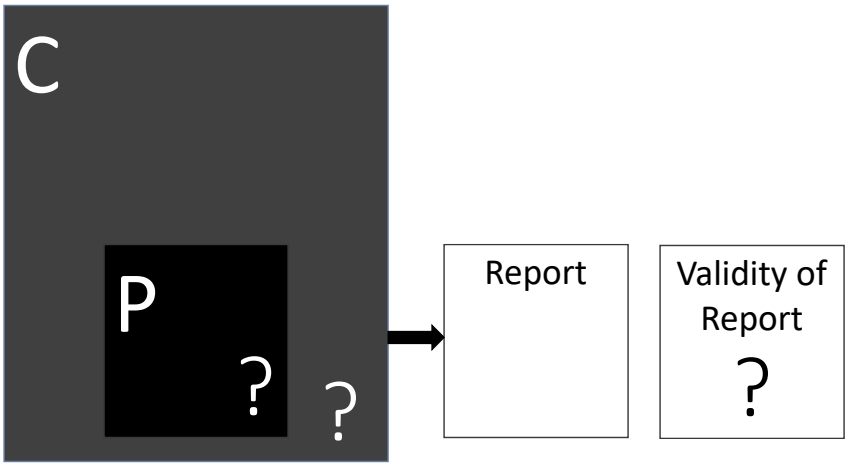
If C is not good, C will incorrectly report that P is good. It may be that P's quality is bad.



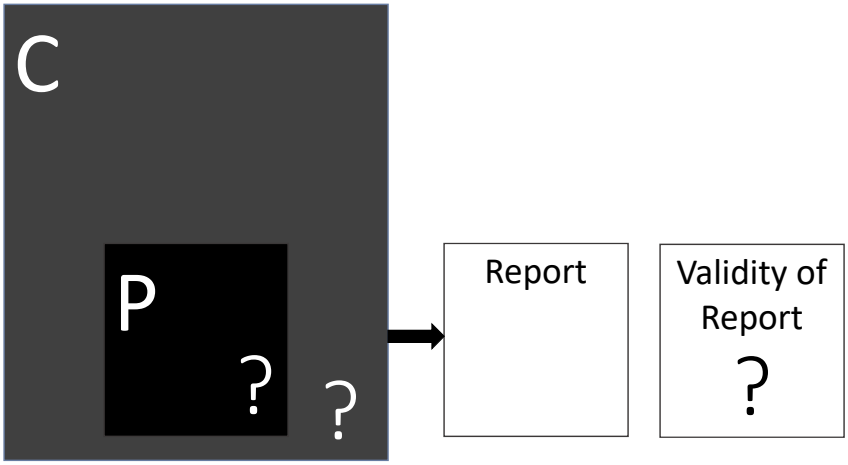
If C is not good, C may incorrectly report that P is bad, even when P's quality is good.



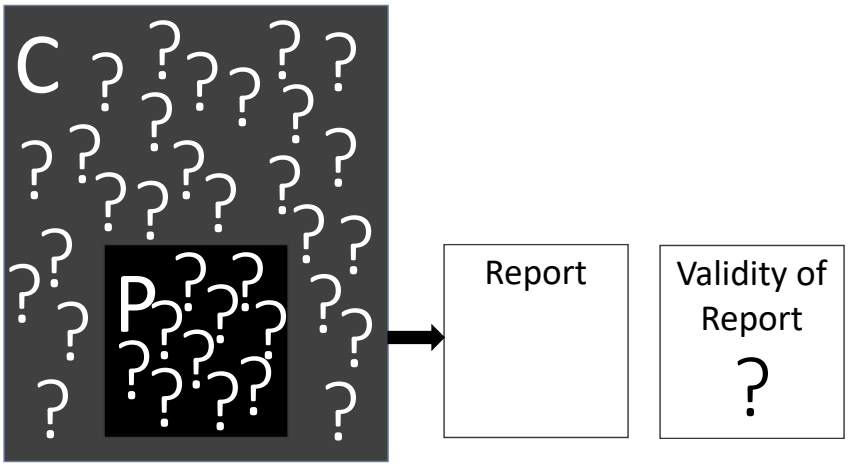
That is: unless we *know* C to be good, we can't be sure about the validity of the report!



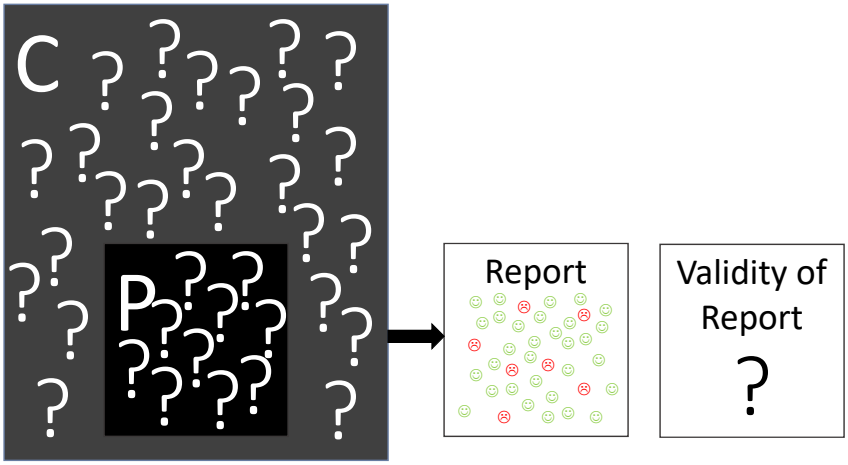
In other words: without investigation and analysis, we can't be sure of the quality of *either* C or P.



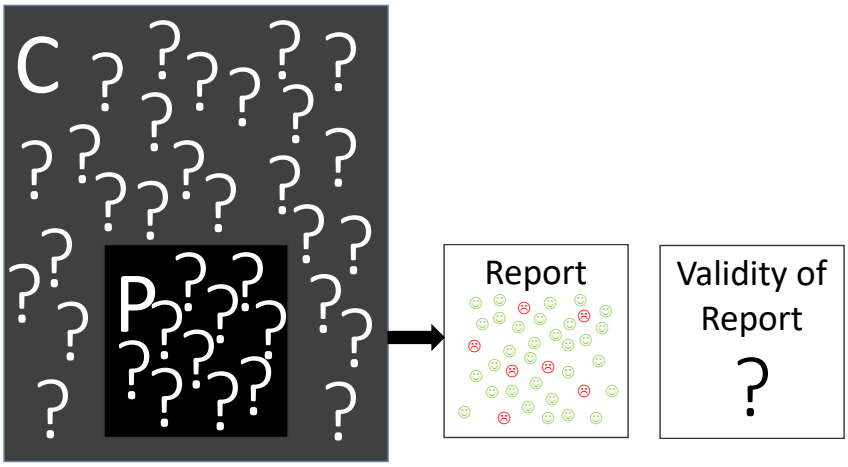
If you think that's bad,
our troubles are only beginning.



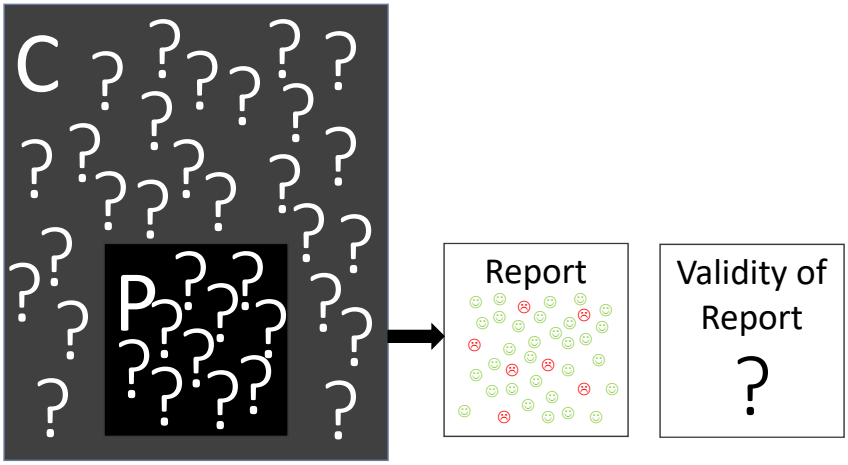
C reports not just a single bit, but on a collection of hundreds or thousands of individual check results.



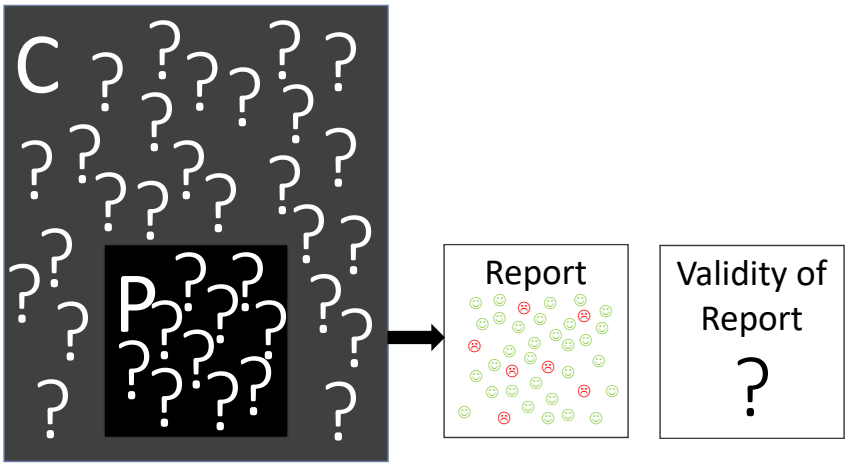
C reports not just a single bit, but on a collection of hundreds or thousands of individual check results.



When a set of checks reports a failure, a responsible tester will not immediately report a bug.



Instead, the tester must investigate and ask if this is a real problem in the product, or a problem with C.

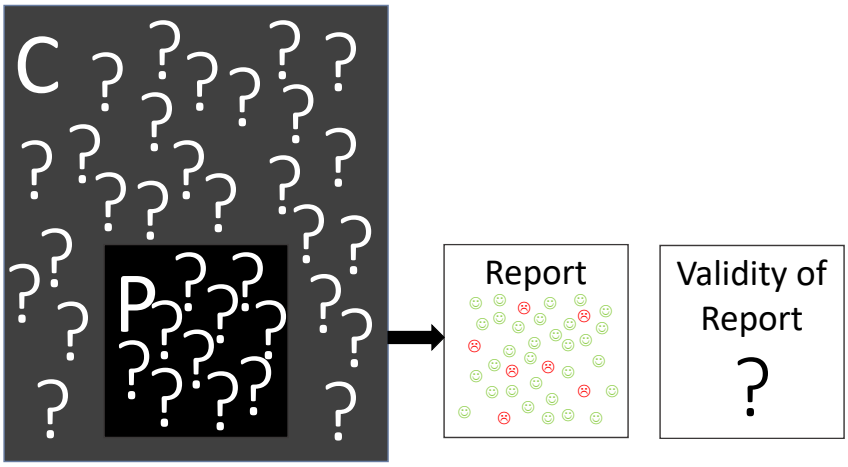


But we've already seen that neither "failing" checks *nor* "passing" ones are always valid.

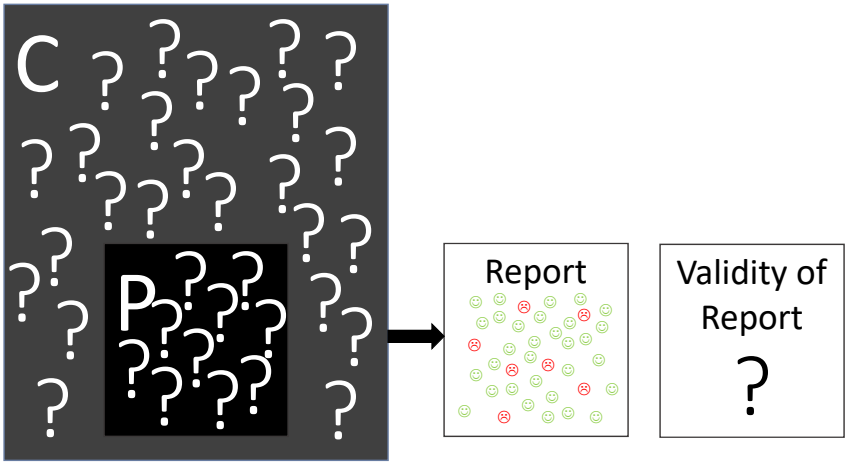
1. A bad product gets categorized as bad because our checks detect problems.
2. A good product gets categorized as not-known-to-be-bad because no problems were found.
3. A good product gets categorized as bad because one or more of our checks is wrong. (Type I error)
4. A bad product gets categorized as not-known-to-be-bad because none of our checks were good enough to detect its particular badness. (Type II Error)

The checks are good if…

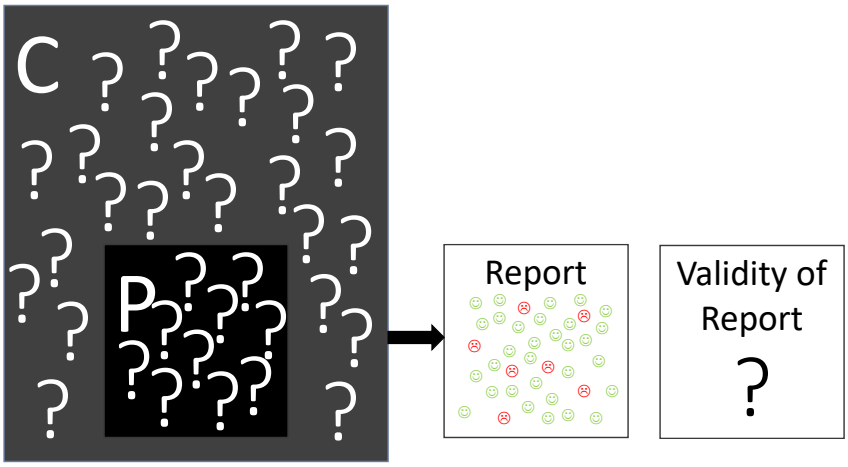
A. We believe 1 and 2 are likely. (validity)
B. We believe 3 and 4 are *un*likely. (reliability)
C. The checks don't cost too much. (utility)



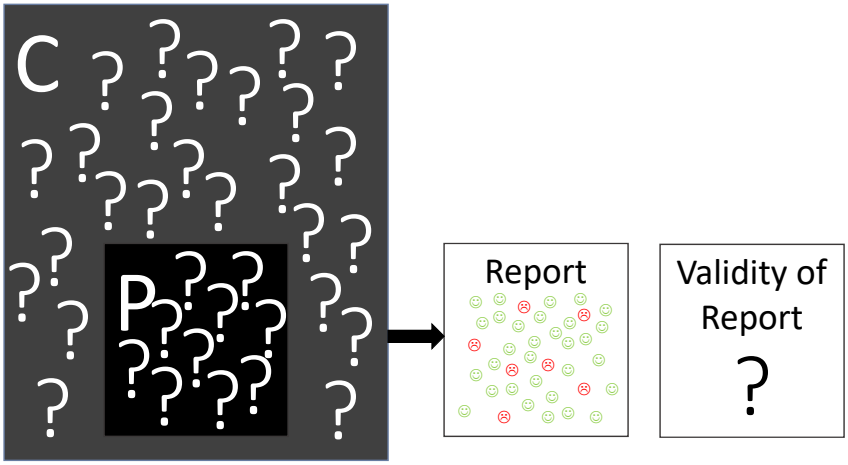
Some people dismiss checks that intermittently report failures as “flaky checks”.



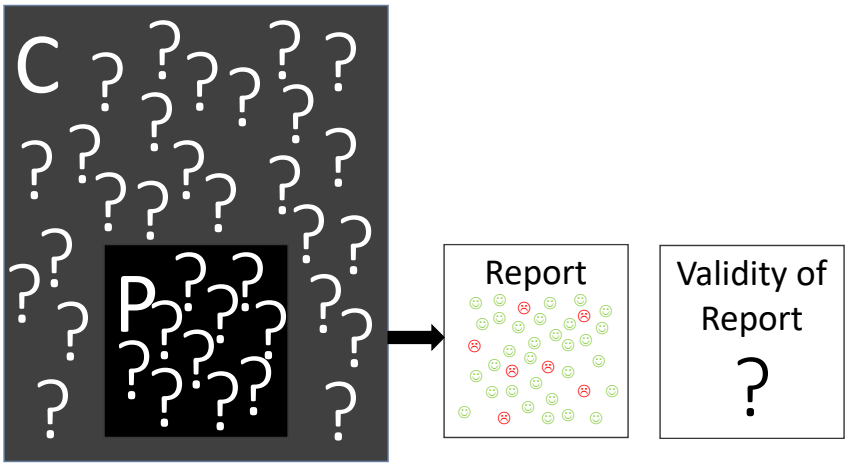
What are we doing to test the idea that “flaky checks” are non-problems?



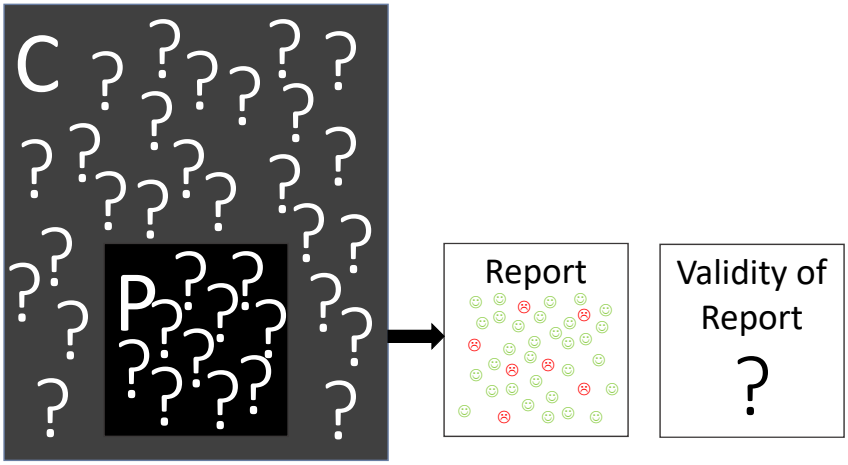
And if checks are consistently “flaky”,
why run them at all?



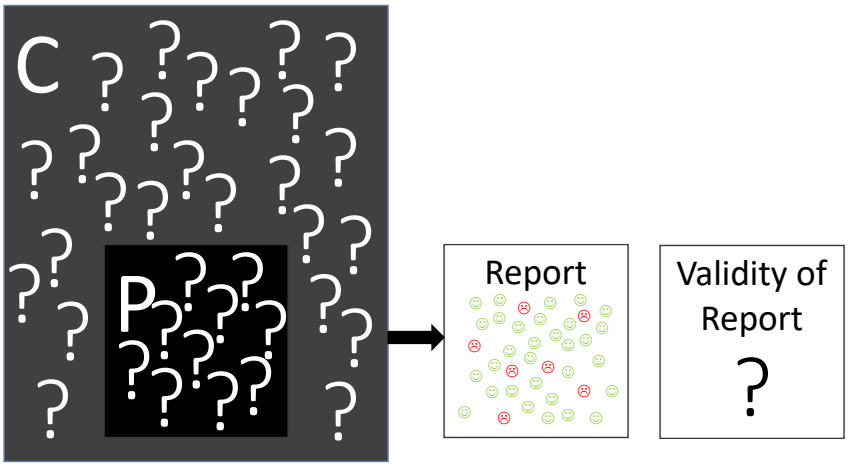
What are we doing to test the idea
that reports of “passing” checks are valid?



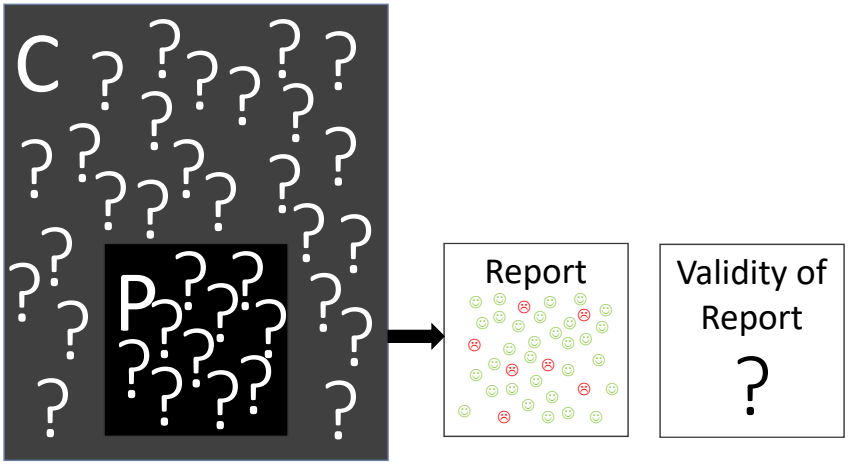
What are we doing to use checks more powerfully—to check more broadly and deeply?



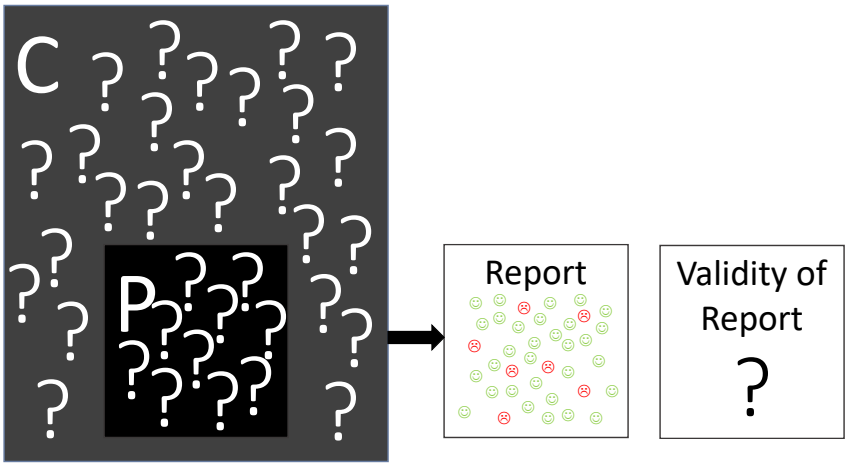
There are many quality criteria that cannot be checked easily: testability, maintainability…



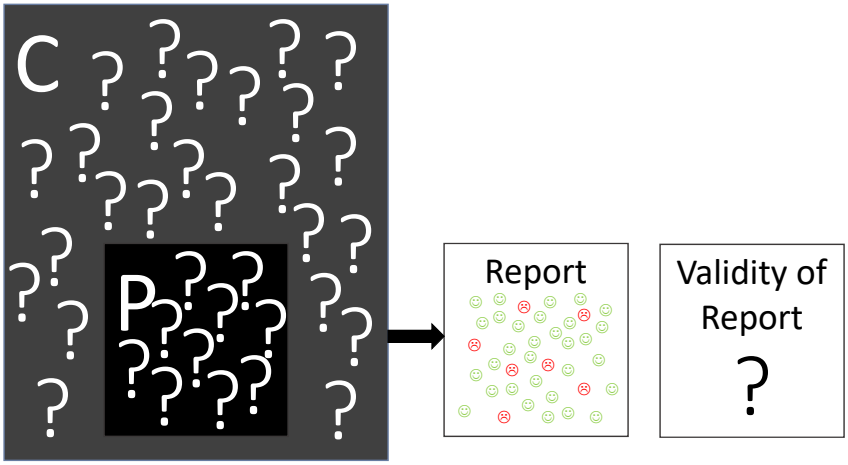
What are we doing to find problems that are not found by automated checking?



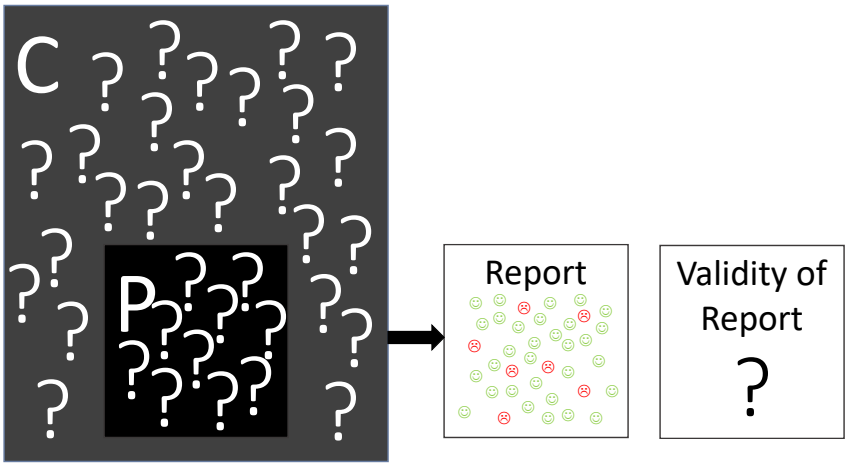
Many people say automated checking allows more time for “exploratory” testing. Is that true?



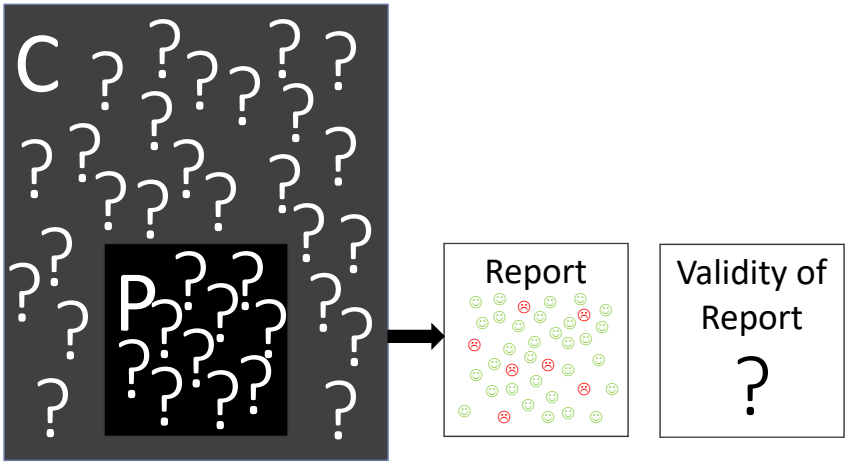
We MIGHT have more time for “exploratory” testing if checks are inexpensive, quick, and easy to prepare.



We MIGHT have more time for “exploratory” testing if we are not investigating too many “failing” checks.



But we might have fewer “failing” checks if we do more “exploratory” testing earlier!



To understand how to do “exploratory” testing before checking, we must learn what testing really is.

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*.

A check can be performed…

by a machine
that *can't* think
(but that is quick and precise)

by a human
who has been told *not* to think
(and who is slow and variable)

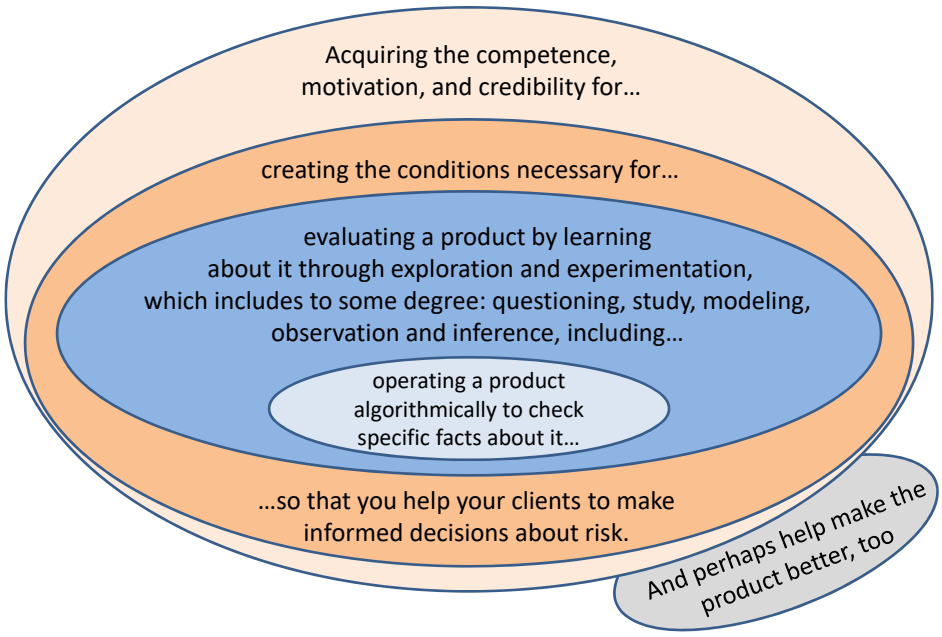
Notice that "quick" and "slow" refer only to the speed of observable behaviours and algorithmic evaluations.
The machine is *infinitely* slow at recognizing unanticipated trouble.

Testing Is *More Than Checking*

- *Checking* is okay, but it is mostly focused on confirming what we know or hope to be true.
- To escape problems with verification, we must do more than checking; we must *test*.

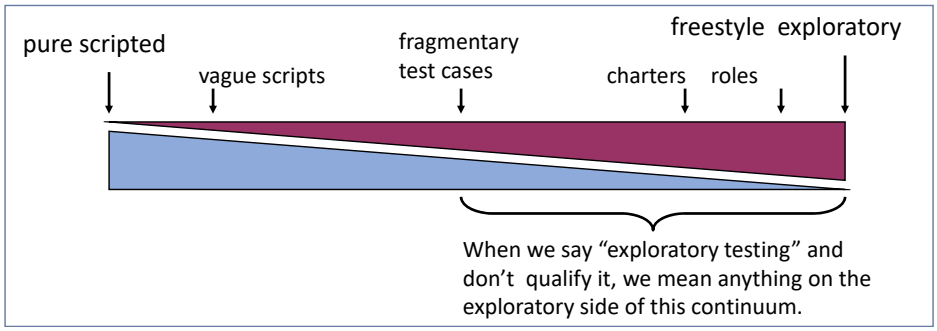
See http://www.developsense.com/2009/08/testing-vs-checking.html

Testing is…



The Scripted/Exploratory Continuum, 2003

- When James Bach was describing testing in 2003, he put scripted testing on the left and exploratory testing on the right. Turns out there was a huge bug in this idea that we didn't notice *for years*.
- It looks like scripted testing comes first! But it doesn't!

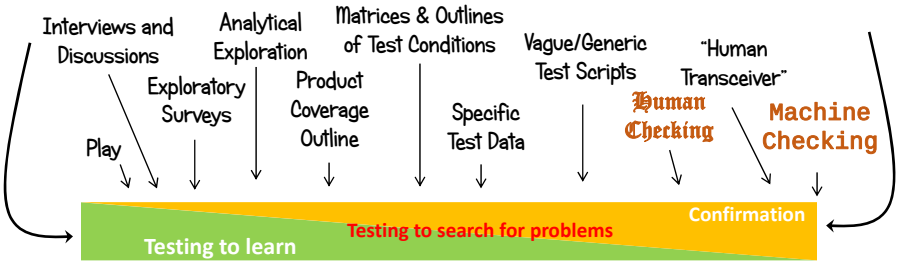


James Bach: The Scripted/Exploratory Continuum from 2003

The *Formality* Continuum, 2014

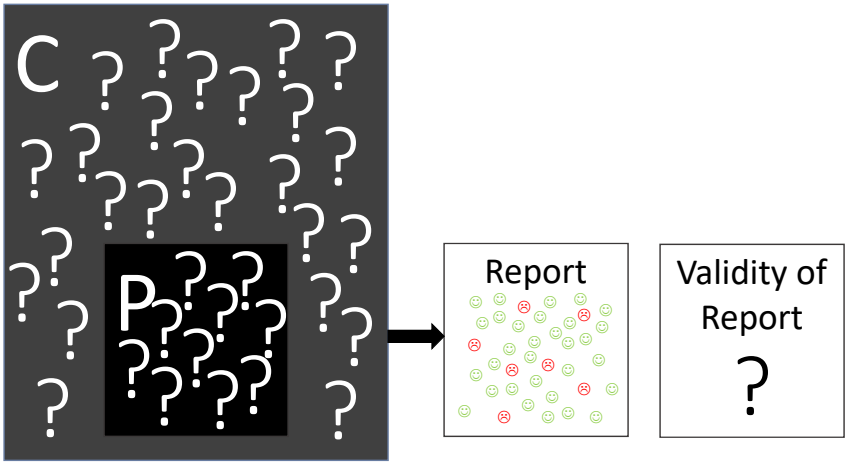
INFORMAL
Not done in any specific way, nor to verify specific facts.

FORMAL
Done in a specific way, or to verify specific facts.



Loops of testing start with informal, exploratory work. If you want to do excellent formal testing (like automated checking), it must begin with excellent informal work.

Bug fix: formality tends to intensify over time, thus showing informality on the left and formality on the right makes more sense.



Now we can talk about what "doing exploratory testing earlier" means.

Exploratory testing includes…

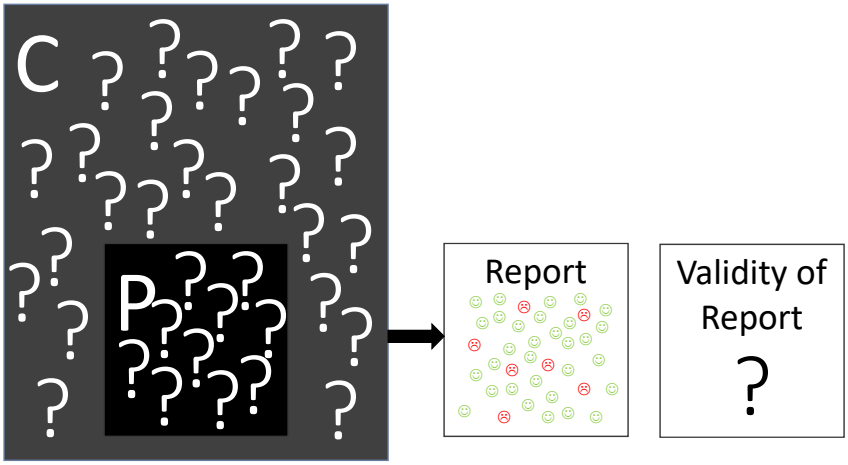
In other words…

Exploratory testing is *testing*.

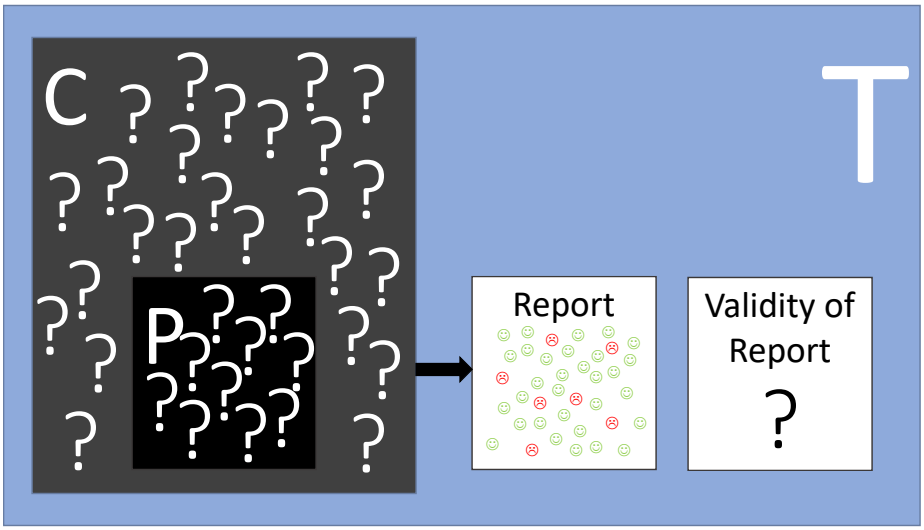
Verification is more like *demonstration*.

You need excellent exploratory work *before* you can do excellent verification.

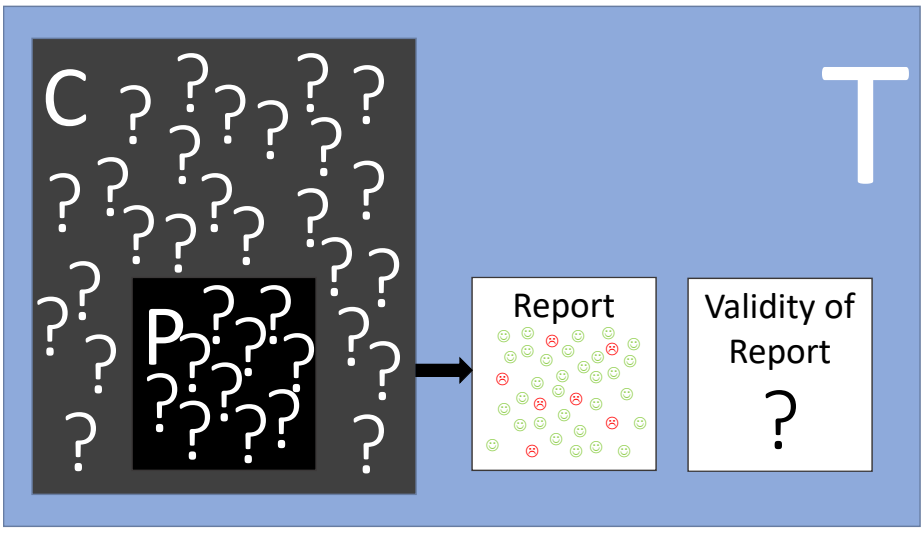
Review
and evaluation
and learning
and sensemaking
and modeling
and studying of the specs
and risk analysis
and recruiting of supporting testers
and observation of the product
and inference-drawing
and questioning
and task prioritization
and coverage analysis
and pattern recognition
and pair development
and decision making
and testability advocacy
and design of the test lab
and preparation of the test lab
and test code development
and tool selection
and making test notes
and preparing simulations
and experimentation
and interacting with developers
and triage
and bug advocacy
and relationship building
and product configuration
and application of oracles
and designing visualizations
and spontaneous playful interaction with the product
and discovery of new information
and preparation of reports for management
and recording of problems
and investigation of problems and working out puzzling situations
and building the test team
and analyzing competitors
and resolving conflicting information
and benchmarking and…



How do we come to a better understanding of the status of the product and the quality of our checks?



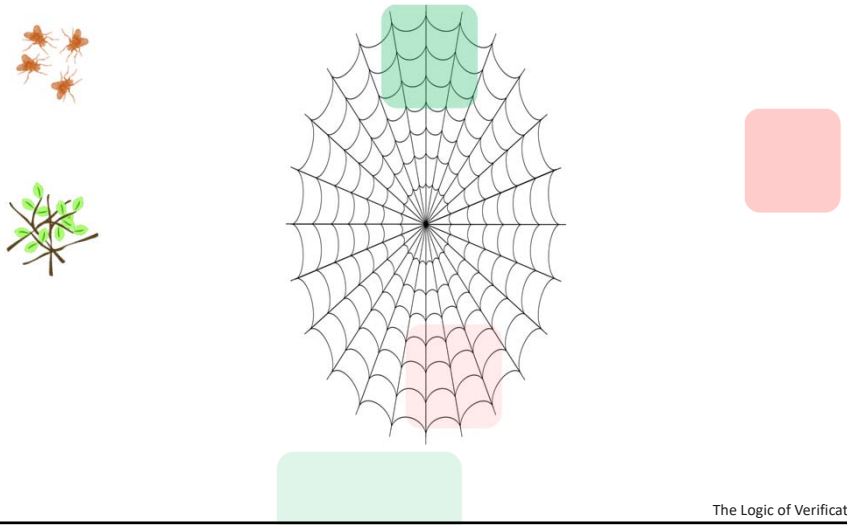
If automated checking is to be valid, reliable, and cost-effective, it MUST be embedded in TESTING.



We must question, study, investigate, observe, diversify and challenge our models, checks, and reports, *and* our ideas about them.

Verifications Form a Web

An *Imperfect* Web



"Researchers are increasingly coming to realise that social spiders also sort themselves according to their individual personalities…

"By paying close attention to individual spiders, [researchers] have discovered that certain spiders are more likely to spend their days attacking predators, while others are more likely to repair the webs, help keep parasites away, clean the web, rear the young, and so on."

http://www.bbc.com/earth/story/20160122-meet-the-spiders-that-have-formed-armies-50000-strong

Way More Than Verification

The Tester is the Spider in the Web

- Testers prepare, supervise, interpret, and maintain **checks and tests**.
- Testers **explore and play and learn** and build mental models of the product and its risks.
- Testers explain and justify their **strategy and status**.
- Testers seek and remove **blinds spots** in test strategy.
- Testers look for ways to refresh and improve the **value of the testing** over time.
- Testers adapt test strategy to the best current knowledge of **product risk**.
- Testers adapt test strategy to the **project context**.

Critical Issues With Most Verification: Poor Sampling, Low Diversity and Weak Oracles.

Workarounds to the Limits of Verification

- Instead of verification, consider *falsification*.
  - We CAN'T verify the idea that the product is okay, but we CAN falsify that idea.
- Instead of validation, consider *assessment*
  - To assess something is to develop opinions on it.
  - You can have opinions about all kinds of things that cannot be verified
  - Our goal is to develop an *informed* opinion of the product.
- Apply safety language
  - "We *have not seen* any bugs *so far*."
  - "We *are not aware* of any problems *yet*."

Conclusions

- To test is not only to verify, but to *investigate* and to *challenge*.
- Excellent testing focuses on exploring and investigating many kinds of risk. Doing this requires many kinds of coverage—not only code coverage.
- Verification (in the form of automated checks or formal scripts performed by humans) may be useful, but it falls short of *testing*.
- Automating checks reduces execution time, but at some cost in development, maintenance, and interpretation.
- Automated checks can be used to test more broadly and more deeply—but there are many more ways to use tools.
- *Excellent* verification is *part* of a testing process that includes not only questioning of the product, but also questioning of the ways in which we check it and test it.

A Word from Our Sponsor (Me)

- I teach Rapid Software Testing (RST).
- RST is a course, a mind-set, and a skill set about how to do excellent software testing in a way that is very fast, inexpensive, credible, and accountable.
- I teach RST in a class for testers.
- I also offer advice and consulting to managers and executives.

http://www.developsense.com