TEST CASES ARE NOT TESTING: TOWARDS A CULTURE OF TEST PERFORMANCE

JAMES BACH
WASHINGTON, USA
AARON HODDER
WELLINGTON, NEW ZEALAND

"We are seeking a Graduate Test Analyst to write and execute test cases to ensure quality is delivered to an extensive client base"

"Responsibilities: Assist with requirements gathering and develop test cases that satisfy the requirements"

"To be successful in this role, the following is required:...To have written manual test cases and involved in test execution...Proven experience in writing Test Conditions & Test Scripts"

"Responsibilities:...Create comprehensive test cases relevant to test conditions...Execute test cases and report results"

From job advertisements listed on seek.co.nz on 24th January, 2014.

OUR INDUSTRY HAS A TROUBLING OBSESSION WITH TEST CASES. WRITING TEST CASES IS mentioned in job descriptions as if it were the main occupation of testers. Testers who approach us for advice too often use phrases like "my test cases" to mean "my work as a tester."

This test case focus has an innocent basis, and is for the most part well-intentioned. But it has become toxic to the field and we, the authors, believe it's one continuing reason why testing commands so little respect. Test cases are holding us back.

It's time for an intervention. In this article, we will critically examine the notion of a test case, and the culture that so often surrounds it. We will show why testing cannot be encoded in test cases, then suggest an alternative vision based on testing as human performance, rather than on artifacts.

> There is no objective, universal way of accounting for test cases

What is a test case?

Definitions vary from place to place. One common idea of a test case is that it is *a set of instructions and/or data for testing some part of a product in some way*. Testers will speak of test cases that are written, or about to be written. In some projects, a test case will always have a unique procedure associated with it. In other situations, test cases may share procedures and be distinguished by unique data.

There is little difference between Agile and non-Agile projects when it comes to test cases. Test cases in Behavior-Driven Development are often specified with a formal structure for execution by tools such as Cucumber. Agile projects tend to be more automation-oriented and more focused on using test cases to define "done."

Here are two simple, contrasting examples:

First, consider a table of test cases characterised by differing conditions. The procedure associated with these cases is implied or documented elsewhere. Thus, test cases may be spoken of as variations on one test idea. Often the contents of each cell are used to report the status of the product with respect to that case. Note that it's not obvious how to delineate or count the cases in this example. "Cookies Accepted," being a distinct idea, may fairly be called a test case, or perhaps each cell in the table is a test case.

	Chrome	Firefox
Cookies Accepted	Pass	Pass
Cookies Not Accepted	Pass	Fail

The second example is also commonly called a test case. It is a step-by-step procedure:

Step	Expected Result	Actual Result
1. Start Chrome browser	Browser starts	Pass
2. Set to accept cookies	Browser accepts cookies	Pass
3. Attempt log in	User home screen displays	Pass

As above, each step might be called a test case because each step has a verification operation associated with it. There is no objective, universal way of accounting for test cases, and hybrids of data-like and procedure-like cases are often found, wherein a procedure-like test

case includes variables that take their values from data-like test cases stored in tables.

These are not the only kinds of things called test cases. A tester can make a loosely structured list of ideas such as “load a corrupted file” and call them test cases. Considering the variety of things called test cases around the industry, a definition that covers all of them would have to be quite general. However, our concern in this article is mostly with detailed, procedural, documented test cases, and the attitudes surrounding that kind of test case.

Note: Often test cases are poorly designed. James was once ordered to create test cases by adding the words “verify that...” in front of the literal text of each requirement. But that silliness is not our complaint in this article. The issues we are raising hold even if you assume that each test case is well-designed. Our claim is that even good test cases cannot comprise or represent good testing.

The Innocent Foundation

Programmers write code. This is, of course, a simplification of what programmers do: modeling, designing, problem-solving, inventing data structures, choosing algorithms. Programming may involve removing or replacing code, or exercising the wisdom of knowing what code not to write. Even so, *programmers write programs*. Thus, the bulk of their work seems tangible. The parallel with testing is obvious: if programmers write explicit source code that manifests working software, perhaps testers write explicit test cases that manifest testing.

“If programmers write explicit source code that manifests working software, perhaps testers write explicit test cases that manifest testing.”

It is seductive to think of written test cases as the “code” of testing. We may covet the sense of accomplishment that comes from producing a tangible asset. We may delight in the simplicity of direct correspondence between test cases and written code. There certainly are situations where thinking in terms of detailed, explicitly specified test cases is appropriate. For instance, when we need to cover a function that can be described cleanly and systematically in terms of a few interacting variables, it can be sensible to model that space formally and then formally specify which points in the space to test. Or perhaps if we want to carry out an intricate and specific test, or a test that requires several testers to coordinate their actions, or even a set of simple fact checks that must be performed periodically - in any of these cases it can help to encode them down step-by-step. And of

> Chocolate-covered french fries have become the staple diet of most of our industry. In too many organizations, testing is fat and slow.

course, if the operation will be performed by a machine, it must be encoded.

But this becomes fertile ground for a vicious cycle: in some specific situations, managers may ask to see test cases for some valid engineering reason, and testers may deliver those cases; but soon providing test cases becomes a habit, and then a tradition, and then someone starts using the term “best practice” as if habitual behavior had won some sort of world championship. The goal of good testing becomes displaced by a blind mandate.

Test cases are not evil. Neither are french fries nor chocolate candy. The problem is the obsession that shoves aside the true business of testing. Chocolate-covered french fries have become the staple diet of most of our industry. In too many organizations, testing is fat and slow. We would like to break the obsession, and return test cases to their rightful place *among* the tools of our craft and not *above* those tools. It’s time to remind ourselves what has always been true: that test cases neither define nor comprise testing itself. Though test cases are an occasionally useful means of supporting testing, the practice of testing does not *require* test cases.

Test Case Culture and the Factory School

Obsession with test cases is not just a habit, though. It is embedded in a culture.

Aaron once ran a small experiment at a software testing course he was attending. He asked fellow attendees whether they write test cases before they start testing. He expected the answers to range from “Yes, of course” to “No” with a healthy dose of “What do you mean by test cases?” thrown in. To his shock, the majority of respondents just looked at him quizzically as if he had just asked them whether they wear clothing to work, or whether they hold their breath while swimming underwater. Test case writing as a central practice appears to go unquestioned in a lot of organisations.

A test case culture is not one that merely encourages using test cases as a tool to support testing. In a test case culture, *the test cases are equated to testing*; testing is viewed as a mechanistic, clerical task of executing test cases (analogous to the mechanistic way that a compiler

turns source code into object code) for the purpose of checking specific and easily observed facts about the product.

A test in a test case culture is a concrete noun, an artifact you can point to and say “that is a test.” Once reified in that way, it is a natural step to treat “tests” as a commodity, like so many sacks of rice. We often see tests counted and testing progress communicated solely in terms of such numbers. Bugs are subject to the same reification; thirteen bugs is one worse than twelve bugs, right? In this culture, bugs and test cases are linked - after all, test cases find the bugs. If a bug is somehow found without a test case (a situation test case culture views with suspicion), we might expect some manager to ask for a new test case to be created that exhibits that bug.

In a test case culture, the tester is merely the medium by which test cases do their work. Consequently, while writing test cases *may* be considered a skilled task, executing them is seen as a task fit for novices (or better yet, robots).

This way of thinking is attractive because it seems to allow testing to be managed with an unambiguous accounting system that makes testers into fungible resources; as a sort of *factory* of testing. Hence we often call this the Factory School of testing thought.

A common attitude about *process* in factories is that there is one right way; and that this right way should be defined and followed. But how that process is discovered is completely outside the scope of factory thinking. In test case culture, this leads to a cartoonishly simplistic understanding of test design. A common phrase in that culture is that we should “derive test cases from requirements” as if the proper test will be immediately obvious to anyone who can read. In test case culture, there is little talk of learning or interpreting. Exploration and tinkering, which characterize so much of the daily experience of engineering and business, are usually invisible to the factory process, and when noticed are considered either a luxury or a lapse of discipline.

“Exploration and tinkering, which characterize so much of the daily experience of engineering and business, are usually invisible to the factory process, and when noticed are considered either a luxury or a lapse of discipline”

A common attitude about *people* in factories is distrust. People are unreliable at following the one right way. People are, at best, a transitional technology: they are tolerated until the right kind of drones can be built. But even in the most successful factories you will notice

that the managers don't consider replacing themselves. At some level, they acknowledge that human attention and action is required. Indeed, here the authors find common ground with the Factory-schoolers that there is such a thing as necessary humanity - except we contend that humanity is necessary at the level where we perform the test.

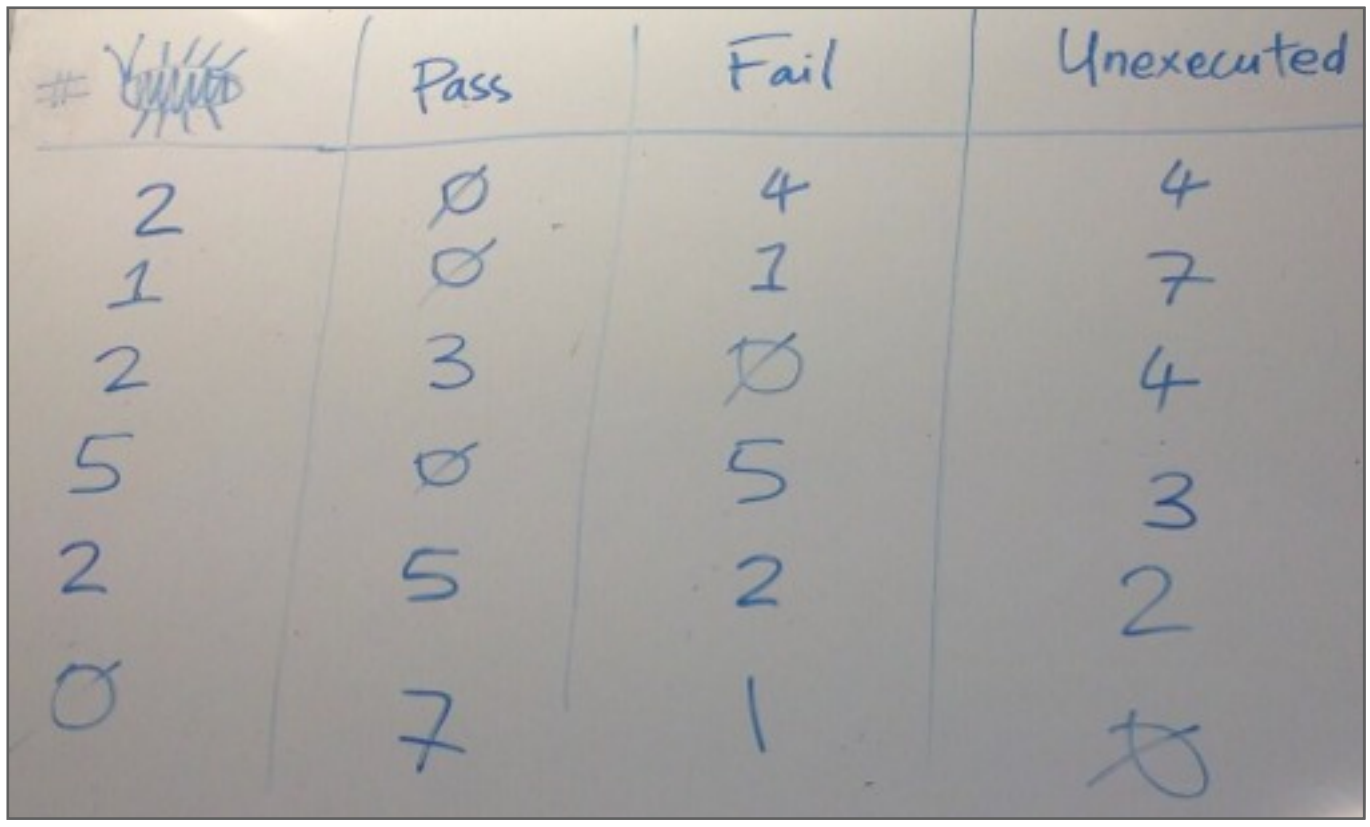
Factory Theory Meets Practice

What happens when we try to manage a complex cognitive activity such as software testing by reifying the activity down to a superficial representation such as test cases?

Recently Aaron observed a class that aimed to teach the limitations of test cases. The students were given eight identical test cases and were instructed to execute them. In Aaron's opinion, these were reasonable examples of relatively good, unambiguous test cases. At the end of the exercise, students were asked for the number of test cases that had passed, the number that failed, the number unexecuted, and the number of bugs found.

One of the purported benefits of a test-case driven approach is that consistency and repeatability are ensured. Aaron expected to see some variation in the results to demonstrate that these claimed benefits are unfounded, but the results were even more striking than anticipated.

Some groups reported no bugs at all, while one group found five. While a few groups reported two bugs, upon further elicitation it was discovered that they weren't necessarily the same two bugs.



This exercise demonstrated beyond even his initial expectations the danger of assuming that test cases can be a reliable medium of conveying testing ideas and reporting on quality-related information. Each group, even though they had the same test cases to work from and were instructed identically, brought their own judgements and heuristics to bear resulting in eight different testing performances.

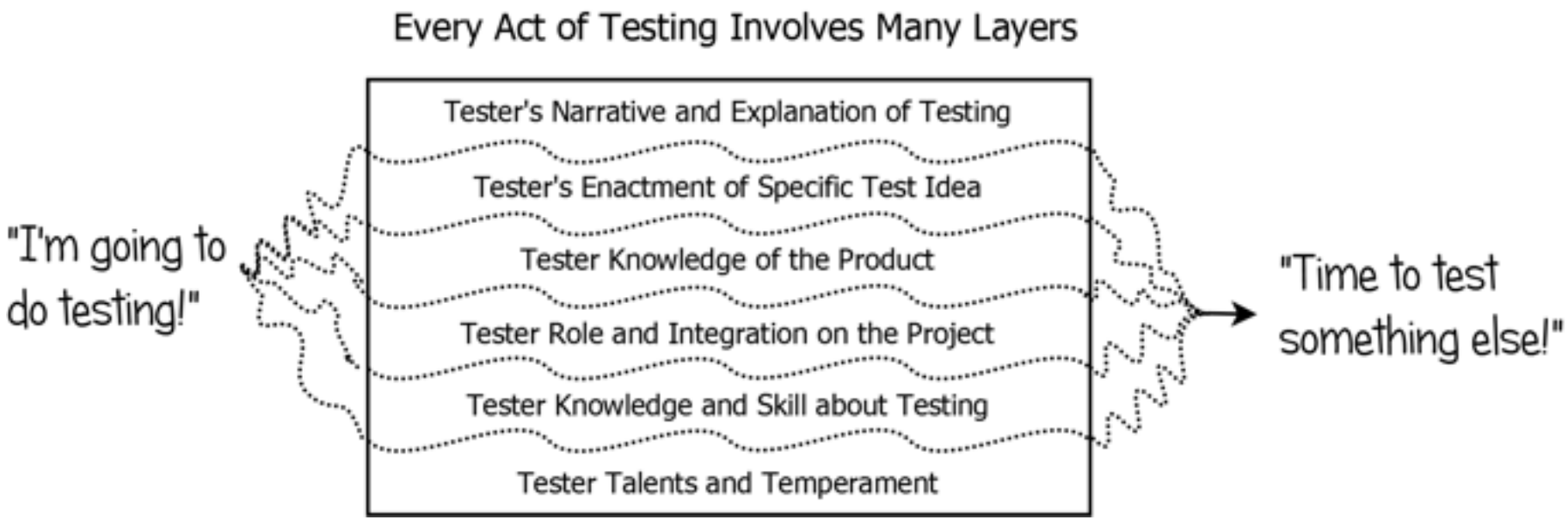
To report on testing by reducing the individual performance down to a numerical report of test cases passed vs. failed is at best, of marginal value, and at worst, potentially misleading. Yet, this is the primary way testing is managed in a lot of organisations.

This case is not a carefully controlled scientific study, but at the very least this experience undermines the glib assurances by many testing consultants and authors, going back to Program Test Methods, the first book on testing, written by William Hetzel in 1973, that written test cases provide a strong and stable basis for testing.

Testing is Not a Factory. Testing is a Performance.

Testing is an event; an activity; a performance. If we work backward from the moment that a tester successfully reports an important problem, we find it results from many overlapping and supporting thoughts and judgments and experiences. There are innumerable ways that this process can play out, all of which require human enactment.

Testing is the evaluation of a product by learning about it through experiment; by seeing it in action. The reason we test is to analyse product risk: the danger that the product will cause trouble for its users or otherwise fail in some way to fulfill its purpose. In other words, we look for anything about the product that might significantly impair its value. We are looking primarily for “bugs.” We want to find every *important* bug, although there will be no way to know for sure that we have succeeded.



Testing as a performance depends on the credibility of the performer, which is improved or damaged in every interaction with the team. But excellent testing is a *complex* performance that is difficult to teach, supervise, or evaluate. While novice testers may find some bugs by romping around like kittens, management needs confidence that there has been a diligent search for important problems. This confidence must come, in large part, from the personal credibility and observable behavior of the responsible tester who vigorously questions the product.

Can an algorithm exist that will guarantee we find all the important bugs? No. This is a matter of basic computing theory (see the Halting Problem) and the fact that bugs are socially constructed by users rather than being something to do with the essence of the product. Bugs are not "in" the product. Bugs are about the relationship *between* the product and the people who desire something from it. It's possible for a bug to be created or resolved just by changing the stakeholder. And apart from every other problem, a total bug identification algorithm would require a complete, unambiguous, and up-to-date specification that is accepted by all stakeholders... and when was the last time you saw one of those?

> "Bugs are not "in" the product. Bugs are about the relationship *between* the product and the people who desire something from it."

When we test we are exploring the relationship between the product and values about the product. To do so, we must make many social judgments, including judging the importance of testing a specific situation, and judging the meaning and importance of potential problems.

This seems hard to accept for many casual observers, though, because many kinds of bugs seem so obvious and uncontroversial. Indeed it is possible to create algorithms to detect *specific kinds* of problems that possess identifiable and predictable characteristics. Many such checks are already built into compilers and application frameworks.

However, even if all imaginable checks are performed, there is no theory, nor metric, nor tool, that can tell us how many important bugs remain. We must test - experiment in an exploratory way - in order to have a chance of finding them. No one can know in advance where the unanticipated bugs will be and therefore what scripts to write. The

performance of testing unfolds forward in time, like a swarm of ants foraging for food. This is a constantly shifting picture.

Although a tester can and should prepare to perform, testing proceeds in ways that cannot be predicted simply because of two things: there is always more testing to do than we can afford to do, and we don't know where the bugs are until we discover each one. This unpredictability requires each tester to be prepared to live in and react to the moment, regardless of any specific plan. That's why, just as flying an airplane, doing surgery, or playing football are rich, complicated performances, testing is too.

> "No one can know in advance where the unanticipated bugs will be and therefore what scripts to write."

Why Testing Cannot be Literally Encoded

Distilling test ideas from tacit mental models; explicitly and precisely describing test cases; writing them down: let's call that "encoding." Encoding means the expression of ideas, explicitly, in the form of some sort of code (such as written words or software). Test case culture insists on encoding testing to such a degree that only trivial aspects of the process should remain undeclared. It insists that encoding is practical and desirable - moreover, that it is *necessary*; that it is irresponsible *not* to encode testing.

But this is not so, because *testing cannot be encoded*.

There is no support in scientific or engineering literature for the idea that testing can be encoded. Despite years of searching, the authors are aware of no studies that have ever shown that testing should be written down, or even that it can be written down. In fact, the opposite is more the case. See the *Sciences of the Artificial*, by Nobel laureate Herbert Simon for a deep treatment of this topic. Simon shows that perfect rationality is unavailable to us in any but the simplest situations, and explores the nature of design processes as "bounded rationality" requiring heuristic solutions. Or perhaps *Introduction to General Systems Thinking*, by Gerald Weinberg, who shows that observing and describing systems requires us to simplify them, and that there is no algorithm for knowing how to do that without losing something that might be important. Or look at *The Social Life of Information*, by Paul Duguid and John Seely Brown, who tell how copy machine repairmen learned their craft not by reading the official documentation but rather by socializing with each other in free-form ways. Or *Things That Make Us Smart*, by Don Norman, who shows how adding a cognitive artifact

(such as a test case document) to a process changes the process in potentially unpredictable ways. If you even *glance* through any of these works, you will see a rejection of mechanistic, reductionistic, algorithmic ways of conceiving and controlling complex systems, including social systems.

In performance terms we consider a *test*, in its noun form, to be *the act of configuring, operating, observing, evaluating some part of a product in the service of a test project*. So, what testers call a test is a process embedded in a larger process called "testing" that includes reading specifications, attending meetings, acquiring equipment, etc. The scope of a test is elastic. It may involve the speculative exploration of a product, or something as simple as checking the result of a function call. There is no objective method by which we can draw sharp lines between individual tests: it is purely a matter of convenience and context how you choose to delineate them.

"There is no objective method by which we can draw sharp lines between individual tests: it is purely a matter of convenience and context how you choose to delineate them."

Testing has many levels, all of which contribute to the success of the testing enterprise, and very little of which can be encoded:

1. **A person is born.** Yes, it's important to start here. Each of us has a specific genetic, environmental, and cultural foundation that means we approach testing with a certain mix of talents and a certain temperament. The fact that James is mathematically inclined leads him to be biased in favor of analytical modeling the things he tests. Other testers may approach the work in a more intuitive or social way. There is no such thing as purely objective and unbiased testing. Two test designers, unlike, say, two car engines, cannot be analyzed and compared in terms of any universal model of testing performance. *Testing talent and temperament cannot be encoded.*

2. **A person learns to test.** Some skills useful in testing are ubiquitous among adults. Others come with general technical or scientific education. Some are technology-specific, and some are specific to testing itself. Learning to test begins in childhood as we play and interact with our world. Testing skill can be acquired in a variety of ways, but deep systematic training in testing is difficult to obtain.

   There is no governing body for the testing field. Therefore, there is no generally accepted Body of Knowledge, or taxonomy of required skills. Commercial certification programs are controversial, but even

if one accepts their view of testing, they do not even attempt to assess practical skill. It is not unusual for people with no significant experience in testing to become a certified tester. This is not possible in reputable fields such as medicine or air transport.

> “It is not unusual for people with no significant experience in testing to become a certified tester.”

The result is that testers vary quite a bit in their practices and their grasp of different aspects of testing. Each tester’s education is local, conditioned by the idiosyncrasies of specific technologies, companies, and projects. And on top of all that, much of testing skill is comprised of inherently tacit skills such as questioning, collaboration, and systems analysis, which cannot be made explicit and mechanical.

No one even attempts to encode the fine details of their own testing skills.

3. **A tester joins a project.** Joining a project is a complex social event. As part of that process, we learn whom we serve. We come to understand the scope of our mission as testers. We commit to the project. This context conditions everything we do as testers. While a mission statement and elements of context can be written down, there are innumerable ways that one might interpret those things and act accordingly. There is no calculus for determining how all that influences testing.

   A tester’s sense of and response to context can be sketched, perhaps, *but not fully encoded*.

4. **A tester learns the product.** Each of us must construct a mental model of the product, its context, and its uses. A more familiar way of saying that is we have to *learn all about it*, and the result of that learning is a mental structure from which we can design tests. This model itself cannot be encoded in any explicit form (it’s neurons, baby). But we can, if we choose, produce some formal and explicit projection of our mental model.

   Therefore while some of our learning can be encoded, most of it will never be, for at least two reasons:

   a. We have no algorithm or mechanism for doing a “brain dump” that accurately reflects the state of our knowledge about anything.

That means we can never rule out the possibility that there is a fact we know about and yet have not put into our model.

b. Even if there were such a mechanism, the totality of what we learn is overwhelming. For instance, to list our true expectations for the behavior of a browser that displays a simple webpage, such as Google's home page, would require prohibitive time and space.

Oh, and that doesn't include an even bigger dynamic: *the process of learning the product is also testing*. The product is operated, observed, and evaluated during the learning process. Therefore, even to the degree that some aspects of testing can be formalized, that can only occur after a substantial amount of un-encodeable learning and exploring work has already been done.

Therefore, the bulk of our product learning, and testing while learning, cannot be encoded.

5. **A tester enacts the testing according to some idea.** This is the act of experimenting on the product, apart from all the processes of preparation that support or inform it.

It is traditional to divide testing into test design, test execution, and result evaluation. Test design may be further partitioned, perhaps into coverage modeling, data modeling, procedure design, oracle design, and tooling. Perhaps this tradition has not served us well in one respect: it seems to imply that there are clear divisions between these activities, and that they are independent of each other. This is not the case. Although when training testers it often helps to focus on each of these in isolation, the practice of testing brings them together in an evolving, exploratory process.

“Although when training testers it often helps to focus on each of these in isolation, the practice of testing brings them together in an evolving, exploratory process.”

This interplay cannot be encoded. The most we can do is write extensive notes about our thinking in every session of testing, but writing it down, beyond a certain point, interferes with testing. And even perfect notes about our thinking would not be an encoding of the *process* of that thinking - in other words we can't write a program, while we are working, that duplicates the workings of our minds.

Picture the process of testing: You look; ponder; try something. You see what happens and ponder *that*. You have a question, then

conceive of an observation that might answer the question. And so on. Skilled testers perform hundreds of what might seem like discrete tests in a session. Few of them need to be repeated. Most tests performed are informed by the results of the previous test. The value of a test may not be known until it has been performed, or possibly much later.

Testing *seems* encodeable because we can crystallize - from out of this thinking-learning-trying soup - acts of configuring, operating, observing and evaluating the product. These moments are embedded in our evolving concept of risk and of the status of the product. But to an outsider, who is not privy to the workings of a tester's mind, they may seem to stand alone. James Bach and Michael Bolton, in their Rapid Testing Methodology, call such acts "checks" if they can be performed, in principle, by a machine (http://www.satisfice.com/blog/archives/856). It's useful to talk about checking because it is a task, embedded within testing, that might be accelerated or substantially supported with automation.

But we must always remember that checking does not represent or comprise testing itself, just as a hammer does not comprise carpentry. We can encode a check, by definition. We cannot encode the process of conceiving, designing, implementing, re-evaluating, or judging the meaning of the results of performing a check.

Some would say we can encode testing simply by recording keystrokes or videoing the test process. Those recordings can be helpful, but they are mere echoes and hints of the testing thought process. They don't encode the richness of the bug seeking and finding intelligence and experience. At no time, when replaying keystrokes, will your test tool stop and say "wait a minute, I think I'm looking at the wrong thing."

Expectations also cannot be encoded. In James' classes, he demonstrates that by asking students to list their expectations for the output of a simple and well-understood feature of an everyday product. Then James proceeds to list dozens of expectations that each student agrees with - but did not think of listing. If testers who try very hard to list expectations can't do it completely even for the simple functions, it is outrageous to think that testing can be fully encoded.

> Test case culture is a ceremonial approach to testing. It is, quite simply, fake testing.

Therefore, with the exception of certain acts of fact checking, enacted tests cannot be encoded.

6. **A tester reports, explains, defends, and amends testing.** Testing doesn't end with the output that the system-under-test produces. The results must be made relevant to the project. This process of reporting bugs and status and concerns happens throughout the test performance process, and it influences that process. It is probably not sensible to separate this process from the operational performance of testing.

   A test report may be partially encoded, but there are innumerable judgments to be made about what to say and what keep silent. Reporting involves responding to questions, too. There is no way to encode an algorithm for that. The act of reporting may spur testers to redo or add to the testing performance, and that also cannot be encoded.

Yes, *bits* of testing can be encoded. We can use encoding to create useful anchor points for the test process. We might use test cases or other kinds of lists, diagrams, or references to formalize parts of testing. These should be considered tools that support testing, not testing itself.

We contend that factories don't apply to testing. While industrial factories are productive (say what you want about how the iPhone is manufactured - you can't deny that it IS manufactured) testing factories do not work. Testing is not manufactured. Testing factories are a big lie. Test case culture is a ceremonial approach to testing. It is, quite simply, fake testing.

But How Can Fake Testing Seem to Work?

Answer #1: *testers may be secretly not faking it*. Brian Osman dubs this practice "stealth testing." This is skilled testing, done for the good of the project, kept hidden due to a management culture that demands performance while at the same time mandating processes that undermine performance. Stealth testing, while well intentioned, helps to perpetuate the test case myth. This is a double-edged sword. If stealth testing finds important problems, and finds them quickly, the tester doesn't get the credit; the approach they actually used doesn't get the credit. The publicly avowed process gets the credit.

Answer #2: *the product may be good enough even with poor testing*. Quality comes mainly from

developers who do a good job. A wasteful test process might amount to little more than a sanity check, and yet the product simply has no important, deep bugs to be found. And face it, many products are pretty bad, and yet still put out there to torment users. The market for software is not an efficient one with respect to quality.

Answer #3: *it is easy to shift the blame for it not working*. Ironically, the inefficiency and ineffectiveness of test factories can be used as an excuse to invest more in them. Once a gullible business has been convinced that testing must be structured in test cases, then any problem that escapes testing seems due to *not having enough test cases*. If you believe in test factories, any problems are either down to the factory not being big enough, or bad people who are sabotaging it. Yet, behind the expensive high walls of test case documentation and the publicly avowed processes that go with them, broken practices of testing can easily hide, and there is little incentive to improve.

Toward a Performance Culture

A performance culture for testing is one that embraces testing as a performance, of course. But it also provides the supportive business infrastructure to make it work. Consider how different this is from the factory model:

- **Testing Concept:** Testing is an activity performed by skilled people. The purpose of testing is to discover important information about the status of the product, so that our clients can make informed decisions about it.
- **Recruitment:** Hire people as testers who demonstrate curiosity, enjoy learning about technology, and are not afraid of confusion or complexity.
- **Diversity:** Foster diversity among testers, in terms of talents, temperaments, and any other potentially relevant factor, in order to maximize testing performance in test teams.
- **Training:** Systematically train testers, both offline and on the job, with ongoing coaching and mentoring.
- **Peer-to-peer learning:** Use peer conferences and informal meetups to build collegial networks and experiment with methods and tools. Occasionally test in group events (e.g. “bug parties”) to foster common understanding about test practices.

> “Testing depends upon testers who have pride and integrity in their work, and who strive to learn their craft.”

- **Openness:** Foster the ability to narrate, explain, and defend testing performances. Create a culture of normalcy about working together and sharing work.

- **Transferring work:** One tester may take over from another with the help of concise documentation, discussion and demonstration, or simply by starting from scratch. Among skilled testers, this is rarely the problem that non-testers fear it will be.

- **Personal Excellence:** Testing depends upon testers who have pride and integrity in their work, and who strive to learn their craft. Part of the reason performance culture is not more accepted in the industry is the lack of trust by management that testers will perform.

- **Team Integration:** Foster a mutually supportive attitude between testers and development. As trust develops, everyone’s performance becomes more fluid and collaborative.

- **Preparation:** Detailed and meticulous planning is rarely cost effective in a high innovation environment such as software development, but that doesn’t mean we can’t benefit from good preparation. Learning about tools and technologies and developing test ideas in concise form is part of performing at our best.

- **Responsiveness:** We recognize that time is of the essence. We look at the product as soon as it is available, and if someone taps us on the shoulder and asks "How did the testing go?" we strive to answer with useful information, confidently and immediately.

- **Cyclic, Exploratory Process:** Performing feeds on itself. When we test, we are also uncovering better test ideas as we go.

- **Agility:** In performance culture, agility is easier, because we aren't traveling with all that baggage of documentation. That means we can respond more rapidly and productively to changing context.

- **Metrics:** Metrics may be used to provoke inquiry, but do not use them as the basis of decision rules to control a social system such as testing. Any metric put in place by management to control people will be used by people to control management.

- **Documentation:** Prefer concise documentation, such as lists and mind-maps, that are less expensive to produce and maintain. Develop a discipline of personal note-taking.

- **Management:** Test leads must supervise junior testers or have them work with senior testers until they are ready to take full responsibility for their own performance. First level management must be involved in testing on a regular basis. Give autonomy to qualified testers to choose their style of work. Celebrate successes, but also celebrate honest, hard-won failures.

- **Process control:** Focus on heuristic rather than algorithmic process controls. Focus on discussion rather than numbers. Focus on trusting people who have earned credibility rather than on inanimate controls and surveillances. If more formal controls are needed, consider using an activity-based approach such as session-based or thread-based test management.

- **Regression Testing:** We may use automated checking tools to help detect obvious problems at each build. These tools are supervised by testers who take responsibility for them. n addition to any checks, however, regression risk may require the tester to enact new tests, or refresh performance of previous tests.

- **Tools:** Use tools under the direction of testers in ways that augment any aspect of tester performance. Do not equate checking tools with human cognition.

- **Stopping:** Testing is finished when the clients of testing feel that every important question about the status of the product has been answered. This feeling is arrived at by discussion of the testing and test results throughout the project.

The Context-Driven testing (CDT) movement, has, for years, been promoting a humanist, performance-oriented vision of testing. There are now two international organizations and several conferences devoted to CDT. Although CDT is not against any practice, it is against methodological chauvinism. The practices we use should be the practices that work well and fit the context. It is through ongoing study and skeptical self-examination that we free ourselves from bad habits and inappropriate practices.

After more than 40 years of trying, the factory approach to testing has not solved the world's testing problems. Enough is enough. Abandon the swamped, lumbering barge of test case culture. Re-discover testing as an intellectual pursuit. The complexities and risks of our world demand that we do this.

The authors thank Michael Bolton and the Testing Trapeze review team for their invaluable review and comments.

“Re-discover testing as an intellectual pursuit.
The complexities and risks of our world demand that we do this.”