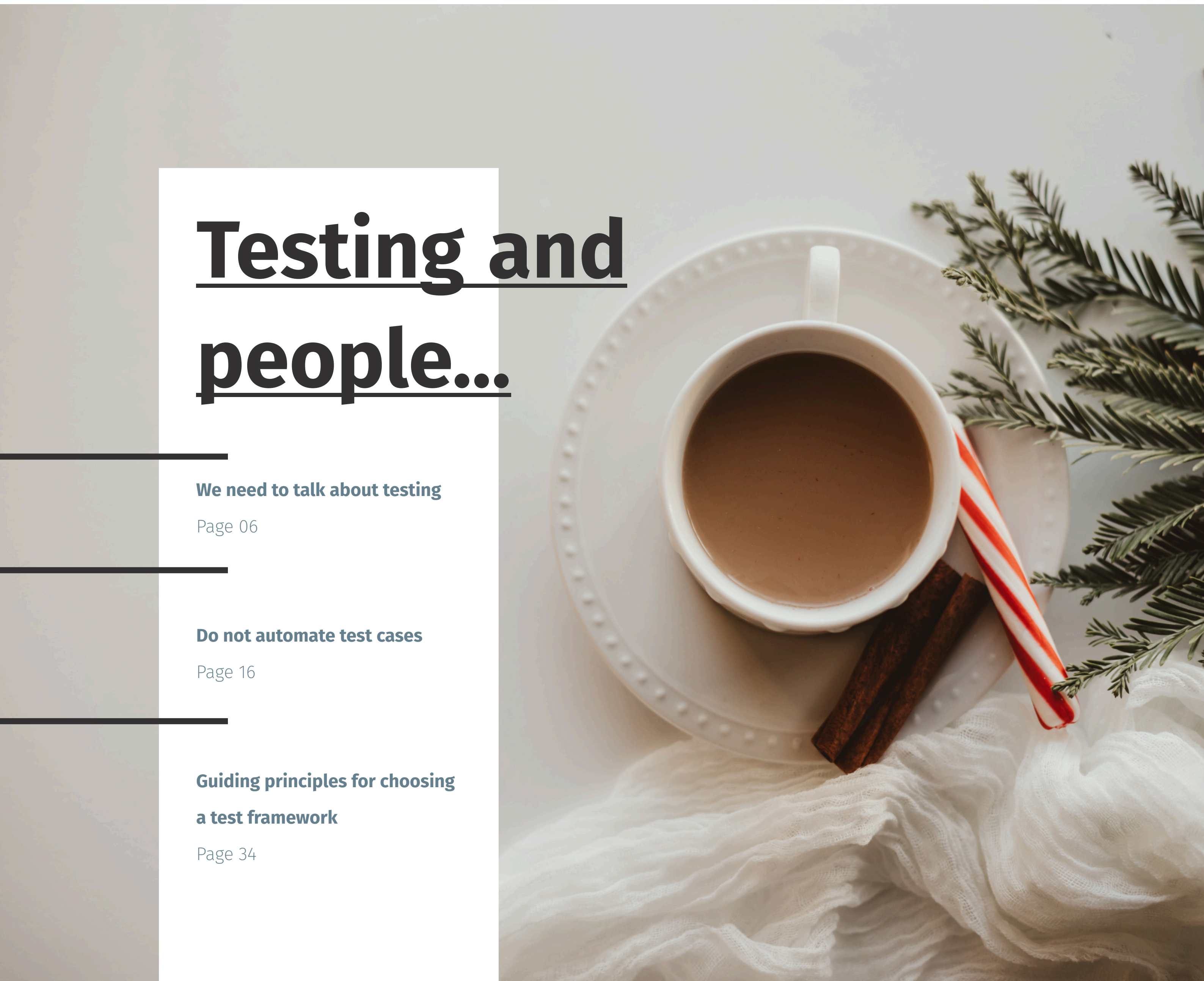
WAKING TESTERS UP SINCE 2011

ISSUE #03/2021

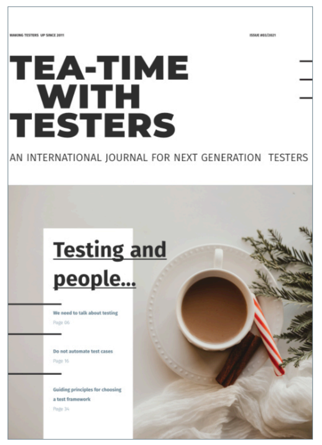
TEA-TIME WITH TESTERS

AN INTERNATIONAL JOURNAL FOR NEXT GENERATION TESTERS



Testing and people...

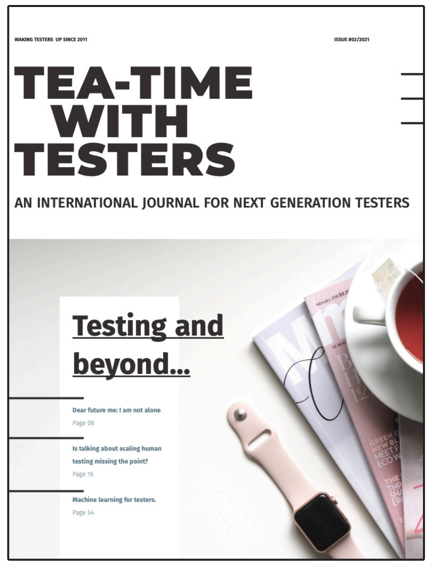
TEA-TIME WITH TESTERS


TEA-TIME WITH TESTERS

AN INTERNATIONAL JOURNAL FOR NEXT GENERATION TESTERS

Testing and people...

TEA-TIME WITH TESTERS



WAKING TESTERS UP SINCE 2011

ISSUE #02/2021

TEA-TIME WITH TESTERS

AN INTERNATIONAL JOURNAL FOR NEXT GENERATION TESTERS

Testing and beyond...

Dear future me: I am not alone

Page 06

Is talking about scaling human testing missing the point?

Page 16

Machine learning for testers.

Page 54

A NEXT GENERATION MAGAZINE

FULL OF CONTENT AND TIPS FOR TESTERS

Brought to you by
Tea-time with Testers
Jerry Weinberg
Testing Excellence Awards
CELEBRATING THE EXCELLENCE IN TESTING.
READ ON TO SEE THE WINNERS FOR 2021

Testing, people and change!

We kept on waiting for things to change, and we hardly realized how an entire year passed just like that.

What an interesting year 2021 has been, right? "What was so interesting about it, Lalit?", you may ask.

Well, when remote working became not optional, I was a bit skeptical about connecting with people and having interactions which I always loved. I was under impression that I was going far away from the people and community but now that I am looking back, the opposite has happened. I presented at multiple conferences across different geographical locations, I got to interact with people from geographies and cultures I was never exposed to before. I got to know about interesting people (and their work) from various disciplines other than software testing.

Oh, and not only that but I accomplished multiple private projects and I finished reading the books I always struggled to find time for. I changed my job and joined an amazing team of very talented colleagues at Accenture Interactive. What a ride this year has been, indeed!

However, there is another kind of change I noticed happening this year and that makes 2021 more interesting for me. The change of perception about testing among testers and non-testers alike has been worth noticing. The rise of various testing tools and frameworks, community forums, conferences, discussions, and debates moved away from blindly automating checks and more towards investing in testing to create value. All of it has been music to my ears.

My personal favorites of the year have been ideas and opinion pieces about testing written by people from different disciplines in software. We are republishing one of such worthy articles in this issue. I would like to thank and congratulate Dan North for penning it down and his deep reflections. Though I do not agree with everything that he says, it is still a worthy article that everyone in the software field should read and ponder upon, especially the non-testing people.

Speaking of people and testing now brings me to our most important announcement of the year. Yes, as promised, we are announcing the winners of the "Jerry Weinberg Testing Excellence Awards" in this issue. We have started these awards to make a difference, to give recognition to worthy testers, and their work. I invite you to join us in congratulating these people and celebrating their contribution to the craft.

Another year has come to an end, and we are excited about 2022 as we'll bring more exciting things for our readers.

On that note, I wish you a great new year 2022! Rock on!

LALITKUMAR BHAMARE
Chief Editor "Tea-time with Testers"
–
Manager - Accenture Interactive
Director - Association for Software Testing
International Keynote speaker.
Software Testing/Quality Coach.

Connect on Twitter @Lalitbhamare or on LinkedIn

WE NEED TO TALK ABOUT TESTING

Or how programmers and testers can work together for a happy and fulfilling life.

Why don't we just automate all the testing? Is test coverage a useful metric? What does it mean to "shift testing left"? When and where should we be testing? How much is enough testing?

Over the years I have discussed these and similar questions many times, with programmers and testers and various other folks. These are important topics and they are often shrouded in confusion, misunderstanding, and tribalism. I have heard from both camps that programmers should / should not be writing tests, are / are not qualified, do / do not even understand testing, and so on.

We usually end up in a better place than where we started, so in this article I want to share some of the discussions we have so that you can have them too.

Much of the confusion stems from a lack of understanding of the purpose of testing, including, ironically, with many testers that I meet, so we don't even have a shared frame of reference.

To create this frame I want to look at a couple of topics, namely:

- What testing is and what it isn't
- Why TDD (and BDD) is only tangentially about testing

From here I will address each of these opening questions and discuss how testers and programmers can collaborate for a happy life.

I hope this will cause you to reassess the discipline and the domain of testing, whatever your role, and to engage with it as the first-class work that it is.

It is a long read, so grab a cup of tea and let's get started.

The purpose of testing

"What could possibly go wrong?"

Whenever we change software — adding a new feature, changing or replacing a feature, making "under-the-hood" changes to improve things — we incur risk. For any change, there is a non-zero likelihood that we cause a Bad Thing to happen.

This is true not only of the code itself but of its build system, its path to deployment, its operating environment, its integration points, and any other direct or indirect dependencies.

There are many types of Bad Things that can happen. Here are a few:

Functional correctness: It doesn't produce the results we expect.

Reliability: It mostly shows correct answers but sometimes it doesn't.

Usability: Sure it works but it is inconsistent and frustrating to use.

Accessibility: Like usability, but exclusionary and probably illegal.

Predictability: It has random spikes in resources such as memory, I/O, or CPU usage, or occasionally hangs for a noticeable amount of time.

Security: It works as designed but it exposes security vulnerabilities.

Compliance: It works but it doesn't handle personal information correctly, say.

Observability: It mostly works, but when it doesn't it is hard to identify why.

For each type of Bad Thing, there is a person or role who cares about that thing, who we call a stakeholder. These people represent the different dimensions of risk, or dimensions of quality, in our endeavours.

Why do we test?

With this in mind, I propose the following statement:

*The purpose of testing is to **increase confidence** for **stakeholders** through **evidence**.*

There are three elements to this statement:

1. Stakeholders are anyone who is affected, directly or indirectly, through the work we do. UX specialist Marc McNeill uses the lovely phrase "people whose lives we touch". This is broader than the customers or end users of a product or service, and stakeholders are more than one-dimensional, siloed individuals; they are collaborators who contribute from different perspectives.

2. Increasing confidence, technically reducing uncertainty, is about understanding the things that the stakeholder cares about, and how the work we are doing — or are about to embark on — might impact those things. How can we help this stakeholder sleep better at night?

3. Evidence is incontrovertible information or data. Stakeholders shouldn't have to depend on your assurance or guarantee, or to rely on your reputation. They deserve cold, hard evidence.

We can describe the process of reducing uncertainty as assurance, and the things that a stakeholder cares about as (their criteria for) quality, so we are talking about quality assurance, which is another popular term for the discipline of testing.1 In our case, we are insisting that we ground this assurance in evidence rather than in blind faith or theatrics.

Consequently, there are three "superpowers" that I associate with a testing mindset:

1. Empathy: the ability to get inside a stakeholder's head, see the world from their perspective, understand their causes for concern.

DANIEL TERHORST-NORTH

–

Daniel Terhorst-North uses his deep technical and operational knowledge to help business and technology leaders to optimise digital product organisations.

He puts people first and finds simple, pragmatic solutions to business and technical problems, often using lean and agile techniques. With thirty years of experience in IT, Daniel is a frequent speaker at technology and business conferences worldwide. The originator of Behaviour-Driven Development (BDD) and Deliberate Discovery, Daniel has published feature articles in numerous software and business publications, and contributed to The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends and 97 Things Every Programmer Should Know: Collective Wisdom from the Experts.

He occasionally blogs at https://dannorth.net/blog.

2. Scepticism: the ability to doubt the work you are doing even while you are doing it. This is especially hard for a programmer: our ego and confirmation bias are always there. This scepticism aligns with the Scientific Method, in which we try to falsify a hypothesis, not to "prove" it.

3. Ingenuity: the ability and determination to do whatever it takes to give that peace of mind to your stakeholder — or to discover they were right to be sceptical in the first place! Testing is non-linear, non-obvious, and often emergent. Poking around in a database; sniffing packets on a network; injecting a service proxy to record interactions; tracking eye movements; hacking DNS; writing code that breaks other code; nothing is off limits to a good tester.

If and only if...

From this definition, it follows that

*You are testing **if and only if** you are increasing confidence for stakeholders through evidence.*

Test thinking includes making architectural or design choices that prevent entire classes of defects, or replacing a hand-rolled UI with a constrained set of robust, hardened, well-documented widgets that have well-understood behavioural characteristics. These are preventative measures that negate certain types of risk: no one downstream has to check for these kinds of errors, because no stakeholders will ever get those unpleasant surprises.

Test thinking means that while you are designing a new feature, you are making time to think about all the different stakeholders who might be interested in this change, and what kind of things they might worry about, and assuming they are probably right.

Conversely, if you are doing work that doesn't manifestly increase confidence for at least one class of stakeholder through tangible evidence then you are not testing, whatever else you may be doing! From a testing perspective, at best you are going over an already-trodden path, at worst you are engaging in test theatre — activities that give the illusion of testing while generating no useful information.

In the context of a digital product, the most common form of surfacing this evidence is hands-on testing, either manually by interacting with the system, or through designing, writing, and running automated tests. Designing and writing good automated tests deserves its own section, but before I get into that we need to talk about TDD, because this is a primary source of confusion.

Test-Driven Development - a sidebar

Test-driven development, or TDD, is a method of programming where the programmer writes small, executable code examples to guide the design of an API or code feature. This approach is a great way to take small, deliberate steps, and to focus only on what is necessary to get the example code working. It tends to produce narrow interfaces and well-structured, navigable code, as long as the programmer remembers to refactor and tidy up as each new example starts to work.

The confusion arises because these code examples can be used post hoc as tests to prevent the programmer from introducing any obvious regressions as the codebase evolves, so early practitioners of this style of example-guided programming called it "test-driven development", and this term entered the mainstream through Kent Beck's seminal agile classic Extreme Programming Explained, and in later writing by him and others.

While they give useful feedback to the programmer, these executable examples don't necessarily make good tests — I will expand on this below — and anyway, experienced TDDers will usually write as few of these as they can get away with! As Kent Beck says, a programmer doesn't get paid to write tests, they get paid to write code that works.

This confusion was my primary motivation for creating Behaviour-Driven Development: I was coaching teams in TDD and we would get mired in all the questions at the top of this article, which it turns out have nothing to do with the practice of TDD. So I formulated a way to describe TDD without using the word "test" at all, and I found that it gave me much better traction and adoption of TDD within teams.

I believe this misuse of terminology, coupled with the belief that most testing is just test theatre, has led the agile movement to inadvertently inflict enormous damage on the discipline of software testing over the last couple of decades.

I appreciate this is a controversial opinion so I will elaborate below. It isn't all negative: the world of testing has benefited enormously from the influence of automated build and deployment; running those "programmer tests" which are better than no feedback at all; working in smaller chunks to drive down delivery time and feedback delay. The culture of "testing in production" typified by emerging movements like chaos engineering are exciting too, but that is still a young discipline.

But the purpose and essence of testing, and the role of its practitioners, has been diluted by the invasive species of "automated tests". Even the infamous automation testing pyramid is usually almost entirely about functionality rather than any of the cross-cutting safety concerns like security or compliance, or about operability or observability characteristics.

To summarise: **TDD, BDD, ATDD**, and related methods categorically do not replace testing, whatever their names may suggest. They are primarily design and development techniques.

So now we know that testing is about providing evidence for stakeholders, and that even if you religiously follow TDD then you are only testing superficially at best, we can return to our opening questions.

"

... the belief that most testing is just test theatre, has led the agile movement to inadvertently inflict enormous damage on the discipline of software testing over the last couple of decades.

Let's talk about testing

Why don't we just automate all the testing?

This is where that enormous inadvertent damage I mentioned manifests itself. Before agile methods came to prominence, during the 1990s, testing was a structured activity that started with test planning, which happened at the beginning of the project, at the same time as the requirements were being written. A key activity at this stage was impact analysis: understanding what was changing and the various ways this might cause Bad Things.

Now I am not advocating to returning to the Dark Ages of software delivery, but I do believe we threw the testing baby out with the big up-front bathwater.

Over the next few years, a new mindset took hold which went something like this:

1. Agile software teams are primarily made up of programmers, with an SME or analyst (later product owner) to provide domain information and set scope.

2. We programmers are using TDD, or "test-first" programming or similar, so we are writing unit tests as we go along.

3. By programming in this way we produce enough tests to be confident in our code.

4. Our unit tests are fast, so we can run them all every time we make a change. We even have a mantra: "Run all the tests, all the time."

5. This means we don't need to do impact analysis for each change. If any functionality is affected anywhere, we will find out as soon as we run our comprehensive unit test suites.

6. We only need testers to mop up after us; to do the testing that is too expensive to automate, or which changes too often to be worth it. We will call this manual testing.

7. As we get better at writing automated tests, we will need fewer and fewer manual testers.

8. Testers will need to retrain as automated testers if they are smart enough — kind of second-class programmers who are only good enough to write automated tests. The others can press buttons. So the hierarchy is programmers > automated testers > manual testers.

9. We can outsource most of this low-value testing and treat it as a commoditised background activity.

10. Our test coverage metrics show how awesome we are at testing. We've got this!

There is so much wrong with this thinking that it is worth taking some time to unpack.

1. [Agile software teams...] Programming is only one part of the value chain of digital product development. The fact that the people in "the team" are mostly programmers masks the wider cast involved directly or indirectly in value creation.

2. [...writing unit tests...] Programmers think they are writing unit tests. They are not; they are writing simple code examples to guide design. Typically these are single-sample tests of functionality, of what the code does, rather than any of the more subtle dimensions of security, accessibility, compliance, etc., unless the feature is specifically for one of those stakeholders, and even then all the others will be sidelined.

These code examples were never designed as tests, but as guides. For instance, as a programmer I may write a "test" with a single value to check a calculation, and maybe one more to triangulate for a more complex case. Then I declare victory. With a testing mindset I will think about edge cases, small or large values, huge negative values, values close to plus/minus zero or other key transitions, missing or invalid values, calling things out of sequence, and various other devious inspections.2

There is no reason a programmer can't adopt this test thinking, but usually we have already moved on to the next thing due to delivery pressures.

3. [...confident in our code...] A programmer's confidence in their own code is a notoriously poor indicator of quality. The person who does the work is by definition the person most likely to be confident in it. If I didn't think it was right, I would have written different code. All kinds of cognitive biases are at play here: confirmation bias, fundamental attribution error, Dunning-Kruger, to name a few.

Navigating all this is the bare minimum for a programmer to start thinking like a tester. We are starting from a position of assumed success rather than assumed failure. My default mode is to seek evidence to "prove myself right" rather than to destroy my illusions, and when I find it, to stop.

4. [...run them all...] This starts out true, but without diligent and disciplined attention to detail, a several-second build becomes a several-minute build becomes a half-hour build. Add the automated provisioning and tear-down of environments, sanitising and loading of test data, contention for build and deployment resources, underpowered development environments, and many other factors, and the dream of "run all the tests all the time" becomes a distant memory.

At that point we have to decide which subsets of tests we run at which stage — and introduce ever more stages to try to shorten intermediate feedback loops while increasing overall lead time to production — and enter the murky world of test suite management.

5. [...impact analysis...] Some tools can take an impact-led approach, by re-running failed tests first, or by using static analysis to figure out which tests may be most useful. But the skill and discipline of impact analysis remains both a necessary and a dying art.

6. [...testers to mop up...] By this stage you can guess where this is going. We need testers to help us think about testing! The axis of automated vs. manual is one of the least interesting to obsess about. Understanding risk and its potential impact along multiple dimensions, and surfacing that all-important information, is a full-time discipline in its own right.

7. [...fewer manual testers...] I tend to agree with this(!). But what an agile programmer thinks of as "writing automated tests" is nothing like the skill and discipline of writing good automated tests. This often comes as a surprise to the programmer.

8. [retrain as automated testers] While programming skills can be useful for a tester, I don't recognise the roles of automated tester or manual tester. Automation is one of many useful tools in a good tester's tool belt.

9. [...outsource...] And herein lies the rub! Recognising risk, understanding impact, getting inside stakeholders' heads — which often involves building relationships with key individuals or groups — and squirrelling around surfacing evidence, these are not activities or capabilities that it is prudent to outsource. If you have an "outsourced testing function", even if it is to another team in the same organisation, the chances are they are engaged in testing theatre, and not doing anything to reduce risk or increase confidence.

10. [...test coverage metrics...] I will talk about test coverage next. We are, sadly, not awesome at testing.

This mindset isn't universal, and many teams have a healthy and comprehensive approach to testing, but it is widespread, and the idea of "manual" vs. "automated" testers is near-ubiquitous, from recruiting to certification to entire career paths.

So to answer the question, why don't we automate all the testing? We should write automated tests when they can help surface evidence, especially for something we are likely to do again and again, and we should do this from a testing perspective rather than a programmer guidepost perspective.

We may also write tools to help with other testing activities, for instance to retrieve test results from a database or remote service, or to preprocess test data into a usable form.

A human being is doing much more than pressing buttons when they interact with a computer system, and the insights and feedback they can produce make hands-on testing a valuable ongoing activity.

Is test coverage a useful metric?

No. Yes. Sometimes. It depends. "Test coverage" is shorthand for "code covered by running automated tests". The multiple quality dimensions thing applies here too. Which stakeholders are we providing evidence for through this particular test? How does this inform their confidence? The fact that some code exercised some other code tells me that at best we gained some evidence for at least one stakeholder.

For example, executing a code path to check for correctness (does it produce the right answers?) tells me nothing about security vulnerabilities, or whether it breaches regulations. And running a test that only checks the same single value every time it runs isn't that assuring in any case.

One thing test coverage can reveal is code that has no automated tests at all! A lack of coverage tells us that code is not being checked by automated tests. But even that is not necessarily a concern if we know that we are verifying the code in other ways. For instance, a user interface that developers and testers see many times each day is unlikely to have a glaring layout error on it, even though there is no automated test to confirm this.

What does it mean to "shift testing left"?

I used to think shifting left meant starting all these testing activities earlier in the process, but I realise it is more than that: it means doing different things. Shifting left on testing means thinking about architecture and design differently, considering different stakeholders early and continually. Which in turn means shifting left on security, accessibility, and all the other dimensions of quality that we should care about. So shifting left on testing motivates all kinds of assurance activities, which can stop us over-investing in a solution that was never going to work. It is like TDD on steroids.

As an unintended consequence, we can remove much of the traditional work that testers would have to do downstream when they only have late sight of the product. Again, we aren't doing that work earlier, we are setting ourselves up to never need it at all!

When and where should we be testing?

This is the corollary to shifting left. The obvious answer is: as early as possible and as often as is practical; wherever in the development cycle we can start surfacing evidence to give assurance. Programmers should be thinking like testers at least some of the time, and testers can help them with this.

There is a fallacy that a feature should be "done" or "fully baked" before we can test it, but this is easily debunked. We can assess what data is being accessed, where from, in what way, for how long, and for what purposes. We can assess lightweight UI sketches for consistency or accessibility.

From this we can have an opinion about security, privacy, compliance. How much data comes back each time? What are some worst-case scenarios for data volumes or values, or screen update times? This can give us insights into reliability, robustness, and resilience.

A programmer can't be thinking like all the stakeholders all the time or they would never get any work done! But this test thinking should be happening continually, with testers embedded in development teams and elsewhere along the value chain.

How much is enough testing?

Any activity that changes a system incurs risk — the possibility of Bad Things — along many dimensions; we saw some of them earlier. Depending on the type of change and the context, the onus is on us to create suitable levels of confidence through evidence.

This suggests that the "amount" of testing isn't a linear quantity. We need to consider "enough" along those same dimensions too. What is the size of the change? What is the potential impact if something bad happens? What are the implications of this for security, usability, and so on? How could we do this differently to change the risk profile?

When we think about "enough" testing, it can lead to constructive discussions about alternatives with different implications. These are almost always trade-offs; it is rare that one solution is objectively "better" along all dimensions than another. Although it is fair to say that a simpler, smaller change is usually less risky than a larger, more complicated one.

Concluding thoughts

As a programmer, I have spent a good chunk of my 30-year career, probably the last two decades, thinking about how testing and testers fit into software development. I have never been comfortable with the positioning of testing and testers in agile software development, and it has taken me a long time to structure and articulate these thoughts. Early hints of this article appear in my BDDx keynote from 2016, and I decided to write it some time in late 2018.

I am not a professional tester. I have written this from the perspective of an agile programmer who these days works within and across teams, and who has had the privilege of meeting and working with some amazing testers over the years.

I believe the purpose and principles of testing can mesh well with agile delivery methods, and the fact that they generally don't is more a historical quirk than a systemic inevitability. If we stay on this path, we will continue side-lining testing and testers, and miss out on the opportunity of genuine high-performing teams.

We can write better quality software faster, and more sustainably, by reintroducing some of these ideas, and enjoy the rare win-win-win of "better, faster, cheaper."

Mentions and Footnotes:

Thanks to Maaike Brinkhof, Aslam Khan, Tom Roden, Anna Urbaniak for their review feedback, and to Liz Keogh and Paul Shepheard for their earlier thoughts.

1. Purists will argue that there is a difference between quality assurance and testing. While researching this article, I read several sources that claimed to differentiate these terms, and each had narrow — and conflicting! — definitions of testing, and used the term "quality assurance" to describe what I am talking about in this article. Many testers I speak with would like to get rid of the term Quality Assurance altogether!

2. There is another style of automated test called a property-based or generative test, where a single test can produce thousands or millions of pseudorandom samples, but that is beyond the scope of this discussion. They are cool though!

Brought to you by

Tea-time with Testers

Jerry Weinberg
Testing Excellence Awards

CELEBRATING

THE EXCELLENCE IN TESTING.

READ ON TO SEE THE WINNERS FOR 2021

TEACHING A SOFTWARE TESTING CLASS - *A FIVE-YEAR UPDATE*

What follows is an update from an article I wrote on teaching a software testing class that was originally published in the May / June 2016 issue of Tea Time With Testers magazine. It's been 5 years since then, and I thought it might be interesting to provide a retrospective along with a peek into what lies ahead. In short, some things have truly changed for the better, but sadly, some things have not. I'll cover both, along with a positive outlook for the future along with a call for your feedback.

Some background. I teach Computer Science courses on a part-time basis at Metropolitan State University located in St. Paul, MN (USA). In my earlier article, I wrote about my experiences teaching my first college level class devoted solely to software testing. On the surface, one might reasonably question the significance of such a class or what makes that experience any different than any teaching other class? I would have thought the same as well, but I faced several interesting challenges that I believe would not have been the case if I were to teach some other area of study, such as programming. databases, networking, and the like.

For starters, senior faculty members expressed sincere concerns that such a course “would never fly.” I was prepared for this and brought to light that testing skills are now a critical expectation for new hires and that effective testing is critical to a successful agile managed initiative. The course was ultimately approved, but I am pleased to report that what began as a one-time special topics course has since been fully integrated into our curriculum as an upper-level elective. I have also been told that plans are underway to integrate more software testing into our other courses. As a final testament of my successes, I have also been asked to teach a graduate level course. I’ll expand on all of this later in the article.

My next challenge was content. In this respect, my goals have not changed. I continue to emphasize the pragmatic, and focus on basic concepts such as soft skills, static testing, black box testing, white box testing, and test planning. I also include a couple of classes on Test Driven Development (TDD) with jUnit and Mockito. Since then, I have been able to incorporate a couple of classes on Behavior Driven Development (BDD) and Cucumber. What is interesting is that software testing is no different than any other area of Computer Science study. Our field is constantly changing, and if we are to succeed as a practice, we need to keep ourselves current. Sadly, it is my experience that many do not, and they resist or remain bitter over the future. Se la vie.

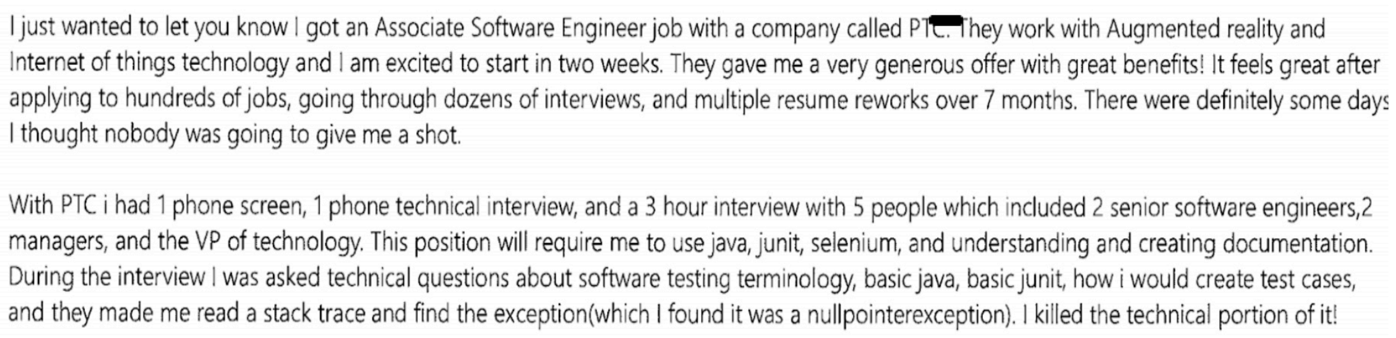
Finally, there was the challenge of selecting an appropriate text. Most computer science courses have an abundance of texts to choose from, but to this day, I have still not found a single college level text that addresses the topics I teach, so I must still resort to several professional grade books. It makes more work for me, but fortunately, the TDD and BDD books come with sample code and exercises.

Metropolitan State is a fully accredited university with a wonderful faculty and solid reputation, but it wasn’t until recently that I realized just how unique it is to offer a software testing class. For my full-time job, I was recently asked to deliver a short presentation on testing to some of our college interns. These were students in their mid to senior years of study, and many of them are pursuing Computer Science degrees. These were very bright individuals, and they were carefully screened and selected from some very prestigious schools. Not one of them knew if their schools offered a course in software testing. I later verified none of their programs do. Worse yet, not one student was even aware that software testing was included in any of their programming courses. Finally, they fared rather poorly when identifying tests cases for the classic triangle problem: http://www.testing-challenges.org/Weinberg-Myers+Triangle+Problem.

Granted, my sample size is limited, but nonetheless alarming. If academia is to prepare its graduates to succeed, software testing needs to be treated with the same level of rigor as other core competencies, and pragmatic and current textbooks are desperately needed. As a practitioner with over 40 years’ experience along with over 15 years’ experience as a part-time educator, I just might be able to make a small but hopefully meaningful contribution. I am envisioning I will get such an opportunity in the upcoming graduate level testing course I mentioned earlier. My plan is to first fast-track most of the content from the undergraduate course. Nothing new here, but it has become apparent that to effectively drive change and raise the bar, the next crop of thought leaders should at least be exposed to systems thinking, change management, and organizational dynamics.

Obviously, there’s enough content in each of these topics to easily consume a course on their own, or even their own area of study, so I must set realistic expectations. Fortunately, thanks to Gerald Weinberg, I have a good foundation to start with. I plan to work in some content from his four volume Software Quality Management series. I will also offer each of these topics as team projects.

I will consider my efforts a success if I can at least raise awareness. Of course, time will tell if I am successful or not, but one of my most personal



I just wanted to let you know I got an Associate Software Engineer job with a company called PTC[redacted] They work with Augmented reality and Internet of things technology and I am excited to start in two weeks. They gave me a very generous offer with great benefits! It feels great after applying to hundreds of jobs, going through dozens of interviews, and multiple resume reworks over 7 months. There were definitely some days I thought nobody was going to give me a shot.

With PTC i had 1 phone screen, 1 phone technical interview, and a 3 hour interview with 5 people which included 2 senior software engineers,2 managers, and the VP of technology. This position will require me to use java, junit, selenium, and understanding and creating documentation. During the interview I was asked technical questions about software testing terminology, basic java, basic junit, how i would create test cases, and they made me read a stack trace and find the exception(which I found it was a nullpointerexception). I killed the technical portion of it!

Other students have told me they introduced BDD practices into their organizations, and others have decided to pursue graduate studies. Yes, this is nirvana, but I call this out to confirm the significance of providing a solid grounding in software testing.

In closing, I am about to retire from a long and rewarding career, but I will be staying on as a part-time contractor, and of course, same for teaching. I’ll end my retrospective with hopes and concerns for the future. Who knows, maybe in another five years I’ll have some exciting news to report. Till then, I’d love to hear from you! If you are a student or recent college graduate, I’d be interested to know how you learned about software testing and what interested you in pursuing it in your career. For you veterans, are you feeling your new hires are adequately trained? If not, what skills do they lack and how are you filling in the gaps? Finally, if there are any educators out there, I’d be interested to know if you currently include or at least plan to include software testing in your coursework and what texts you are using.

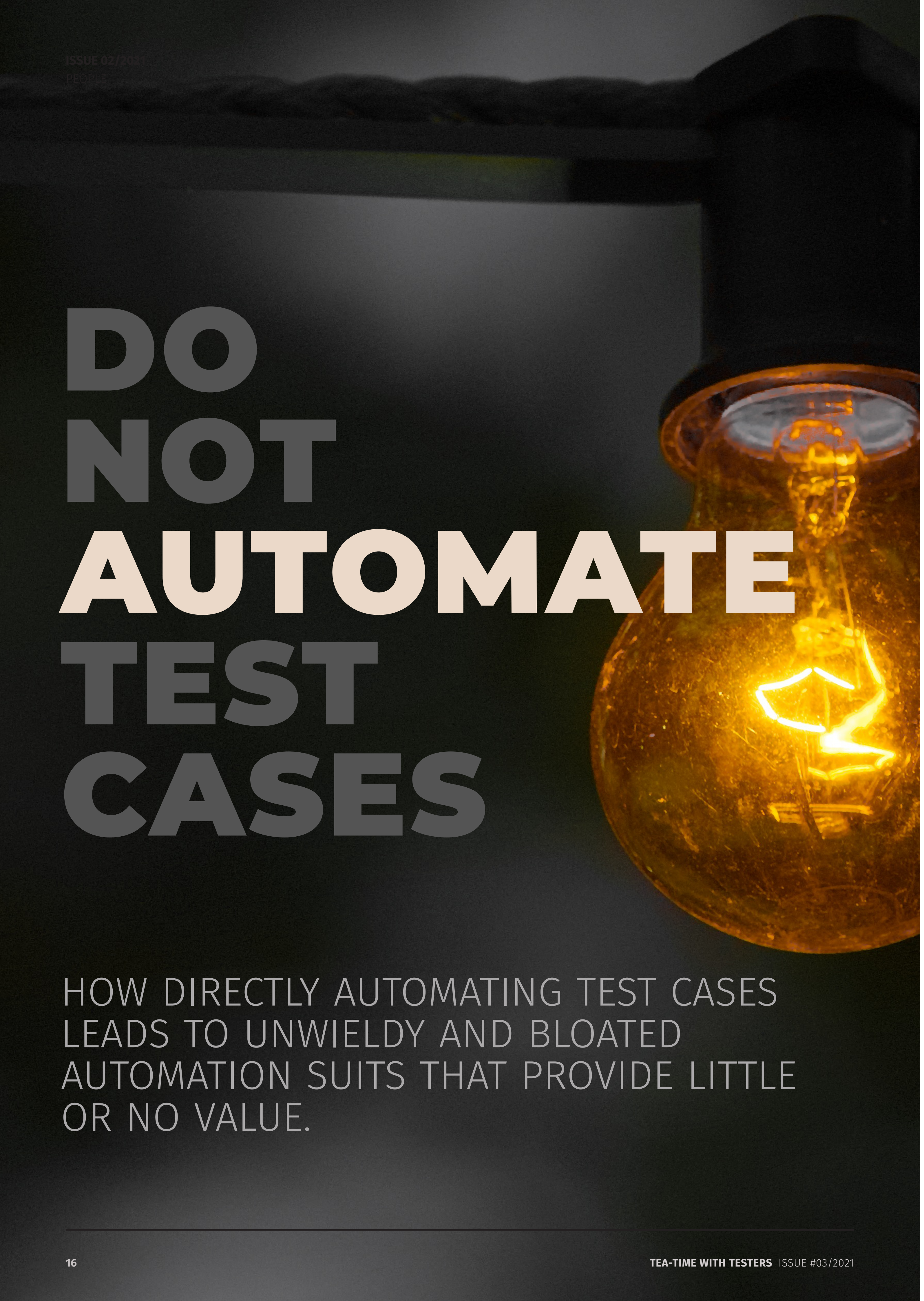
DAVID LEVITT

–

Mr. Levitt is a passionate software engineer and educator. He’s held lead roles as a programmer and tester, and he has successful track record of driving change – at least some of the time. He holds a BS and MS in Computer Science and an Advanced Certificate in Software Engineering.

He can be reached via LinkedIn - Dave Levitt or david.levitt@metrostate.edu

Call for articles
It’s the right time to write for Tea-time with Testers.
Email: editor@teatimewithtesters.com
www.teatimewithtesters.com



DO NOT AUTOMATE TEST CASES

HOW DIRECTLY AUTOMATING TEST CASES LEADS TO UNWIELDY AND BLOATED AUTOMATION SUITS THAT PROVIDE LITTLE OR NO VALUE.

It is a common practice to use test cases as a backlog for test automation. QAs develop test cases from user stories as part of normal testing, then automate those tests. Each iteration, more stories are tested, more test cases are automated, and the suite of automated tests grows larger. Engineering leaders push metrics like "percentage of test cases automated" and congratulate people on that high number. Some teams even employ specialized "automation engineers" whose sole job is to take test cases and automate them.

Unfortunately, automating test cases and pushing percentage-of-tests-automated metrics is a quality engineering anti-pattern and inevitably leads to bloated and hard to maintain test suites that provide little value. While automation is critical for agile delivery, this overly simplistic "automation factory" mentality is not a healthy approach to test automation.

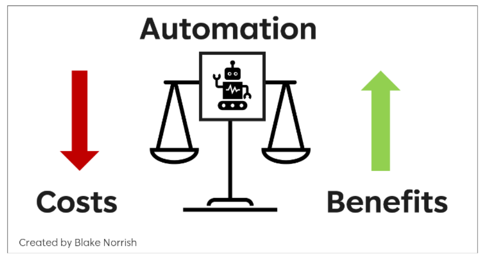
In this article we will show why automation factories are misguided, and describe a better approach to automation development that ensures test automation sustains and accelerates delivery velocity.

The Costs and Benefits of Test Automation

To understand why automating existing test cases is so problematic, we need to step back and review a bit of automation theory. Specifically, we need to dig into the automation costs and benefits, look at the expected value-over-time of automated tests, then look at how the expected value changes with different types of tests. We can then look at how automating test cases using a simple automation factory approach would impact overall test suites.

All automated tests have two types of costs: an initial cost to develop, and the ongoing cost of maintenance. Tests are expected to have some benefit: the difference between the time to execute the test manually, and the (assumed) much quicker automated check. While there are other intangible benefits (it's more fun, it teaches valuable skills, etc.) we do not need to consider those here.

While this is a massive oversimplification of test automation, it does capture the critical aspects for our purposes — every automated test of every type has both a cost and a benefit, and both are important. As automation experts we are trying to maximize the benefits and minimize *the costs.*



The most oversimplified analysis of test automation you will ever see.

Here are some things that affect the cost of the test:

- The existence (or lack) of a test framework into which the test will be added
- The cleanliness of the existing test framework and suite
- The ease of and ability to setup test state (eg test data)
- The availability of an acceptable test oracle
- The volatility of the interfaces or features the test will interact with
- The stability of the environment against which the test will run
- The technical skill of the QAs expected to create and maintain the automation

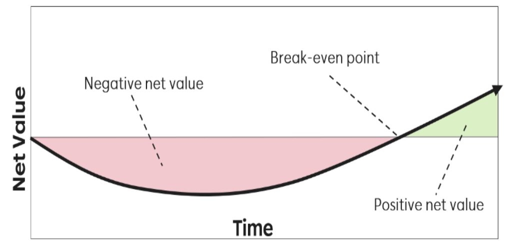
Here is a partial list of things that could affect the amount of *benefit* we derive from an automated test:

- How often the test is expected to run (every commit, daily, per release, etc.)
- How long the test will be a valid check of the SUT.
- How expensive (in time, or otherwise) it is to validate the same test manually
- How error prone the test is to run manually

Both these lists are incomplete, and experienced quality engineers are probably screaming "But what about... xyz!!" Fortunately, an exhaustive list isn't necessary to show that every test has a litany of factors that contribute to both the expected costs and benefits.

We've already established that automation costs can be split between upfront development and ongoing maintenance costs. The benefits also have a time dimension: value from the automated test isn't realized immediately, it's accumulated over the lifespan of the test.

So the full account of the value isn't final when it's created, but rather something that changes over time. If we graphed this value-over-time for a generic automated test, it would look something like this:



BLAKE NORRISH

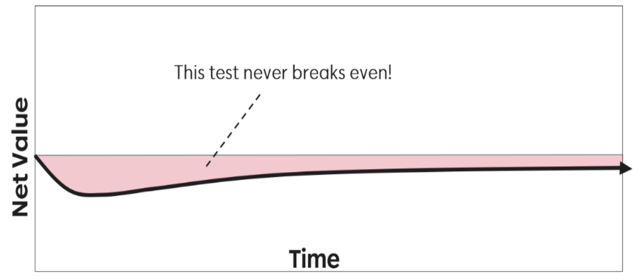
–

Blake Norrish has been building and testing software for his entire professional career. After twelve years as a test architect at Expedia and five years consulting at Thoughtworks, he joined Slalom Build where he is now Senior Director of Quality Engineering.

Follow him on Medium or connect on LinkedIn.

This graph shows a test that initially has a net-negative value: the initial development costs outweigh the benefits in time saved. However, the time saved eventually overcomes the initial cost as well as ongoing maintenance costs to provide positive value. Every test goes through some lifecycle like this based on all the variables listed above.

Depending on initial development costs, maintenance costs, and benefits derived, the test may break even and provide positive value much sooner, or possibly never break even:



The above graph describes a test that was initially providing some value (after initial development, the line slopes up), but then later stops providing value. Perhaps this test stopped being run, or was so flaky that nobody trusted it, or the feature it tested was sunset. Regardless of the reasons, the graph tells us that we would have been better *off never automating the test in the first place.*

This is the key point: that the value of automated tests is impacted by many variables, and depending on these variables, automated tests can have either a positive or negative lifetime value in terms of overall time saved.

The Many Types of Automated Tests

Next, let's consider what the value-over-time graphs would look like for different types of automated tests. By types of tests, we mean everything from small, code-level unit tests, slightly larger "social" unit tests, even larger component tests, higher level integration tests, tests that bypass the UI and hit APIs directly, tests that mock the APIs and exercise just the UI, E2E tests that span the full tech stack, etc.

It's important to note that in complex, modern software there are many, many possible types of automated tests; there are an almost unlimited number of ways you can decompose the system into pieces and attempt to isolate just this piece or that piece. Each of these isolations represents a possible test type.

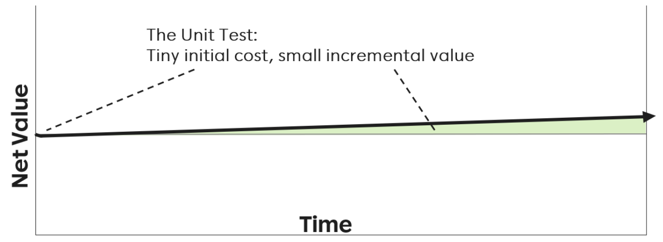
An exploration of every type of automated test that could conceivably be created against a non-trivial architecture is beyond the scope of this article, but I'd recommend the Practical Test Pyramid and Test Strategies for a Microservice Architecture as good primers. To keep this post succinct, we will only consider the two extremes: the small unit test and the large, full end-to-end (E2E) test.

What would the cost/benefit equation and the graph of the value-over-time look like for a unit test? Some unique characteristics of unit tests:

- Unit tests can usually be written in a matter of minutes, if not seconds
- Unit tests are (or should be) immune to external state. Meaning they set up mocks, doubles, stubs, etc. to deterministically control the execution path.
- Unit tests can be executed in milliseconds, suites of unit tests in seconds.
- Unit tests will likely be executed thousands of times a day, not only within a CI/CD pipeline for every commit, but also locally by each developer as code is written.
- Even with 100% coverage, unit tests cannot prove the application actually works as expected, and only validate an incredibly small (usually singular) thing.

Given this assessment, the value-over-time graph of a unit test is probably very different from our generic graph: it has almost no up-front cost, minimal maintenance, and while it is executed all the time, each incremental execution actually provides only a very small amount of value.

The graph would probably look something like:

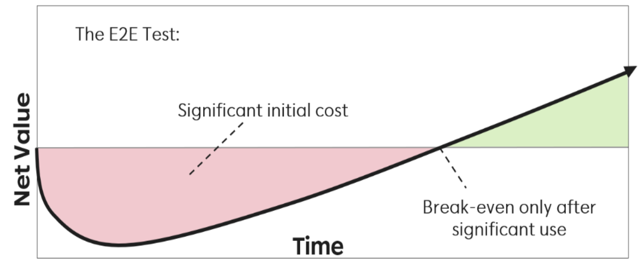


What about the largest of all automated tests, the E2E? What would the value-over-time graph look like for it?

Some key points about E2E tests:

- E2E tests (by definition) are impacted by the most state and thus require the most setup and test data control.
- E2E tests are executed against a full environment. Often parts of this environment are shared.
- E2E tests often include many (dozens, even hundreds) of serial steps.
- E2E tests are the slowest of all tests, possibly running for minutes.
- E2E tests usually have to drive functionality through a user interface.
- E2E tests are usually executed much later in a CI/CD pipeline.
- E2E tests are the only type of automated tests that demonstrate the application works as the customer would use it.

Given these points, the value-over-time graph of the E2E would be something like this:



This graph shows a significantly higher up-front cost, initially sending the net value highly negative. However, the continued execution of this test over time eventually allows it to break even, then proceed into positive value.

Again, the expectation that this test will eventually provide positive value is predicated on the assumptions made at the top of this section, things like: how long the test will be used, how often the test will be run, how much confidence we have in the result of the test, how much the test will have to change when underlying interfaces (like the UI) change, how stable the environment we are executing against is, etc. Reaching break-even is never guaranteed.

Where to Automate

The nature of E2E tests makes them costly to create and costly to maintain. They necessarily rely on (or could be impacted by) the most states across the most systems. They are more prone to system timing, synchronization, network, or external dependency issues. They usually drive some or all functionality through a web browser, an interface designed to be consumed by a human, not software. Because they are executed against a full environment, it is more likely that these tests will have to share part or all of this environment with other tests or users, possibly leading to collisions and unexpected results.

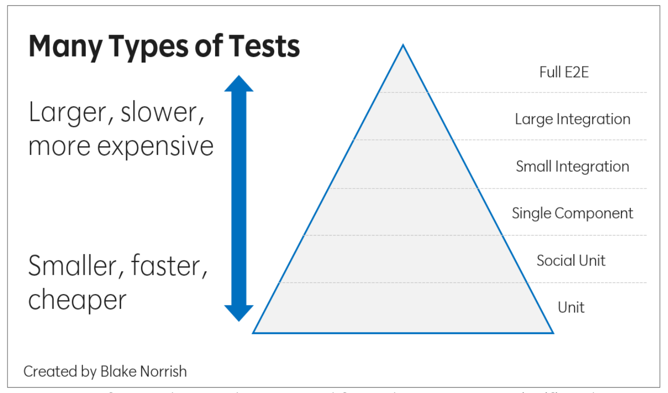
All these reasons (and many more) make the large tests most risky, and are only justified because of the complementary large value they can provide: only E2E tests demonstrate the full integrated system working together in a realistic manner.

There are many other types of tests to consider, and within complex systems, it's highly likely that a lower level type of test could more directly test the functionality in question without incurring the costs associated with higher level (larger) tests.

In other words: do not build an API test against a deployed service instance to validate a piece of logic that could be validated in a unit test. Do not build an E2E test for logic that could be validated by directly calling a single API. Never bring in more of the system than you have to, to demonstrate that something is working. Identify the logic or behavior you need to test and create a test that isolates exactly that behavior. Only use higher level tests to test the actual integration of things, not the logic within those things.

The huge, overarching point is that E2E tests are risky and are rarely the best type of automation to test specific functionality. To put this in terms of the value-over-time graph: always prefer test types that give you the most immediate expected value with the least cost, and be skeptical of large automation efforts only justified by overly optimistic estimations of eventual time-saving benefits.

This type of cost-benefit analysis is exactly the thinking that led to the Automated Test Pyramid concept many years ago. The pyramid advocates that, all things being equal, you generally want much more small-fast-cheap tests and much less large-slow-expensive tests.



Many types of tests. The actual names used for each test type vary significantly between organizations.

While I won't try to convince you that the shape of your suite must always and exactly be a pyramid (Kent Dodds says it's a Trophy, James Bach prefers a Round Earth model, Justin Searls says it's just a distraction), I hope I did convince you that all test automation carries risk, and ensuring tests are created at the appropriate level helps mitigate and control this risk. A huge part of the automator's job (in collaboration with the rest of the team!) is determining exactly which type of test is appropriate and gives the highest likelihood of lifetime positive value.

Said in a different way: all automation should be treated as an *investment*, specifically, a *risky investment.* Each type of automated test represents a different type of risk, and we need to manage overall risk and maximize the value of our investment by continually evaluating the cost and benefits of each type.

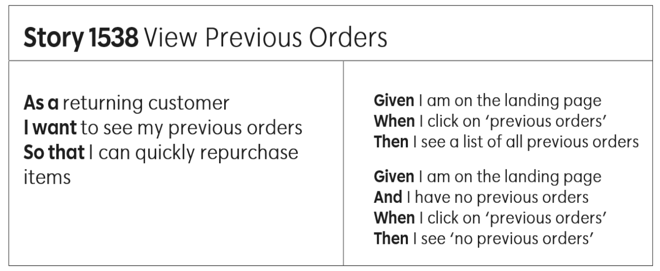
All automation should be treated as an investment, specifically, a risky investment. Each type of automated test represents a different type of risk, and we need to manage overall risk and maximize the value of our investment by continually evaluating the cost and benefits of each type.

Automation Factory and the Top Heavy Suite

Back to the automation factory!

If we think about user stories, acceptance criteria, and the test cases that a quality assurance professional would create from them, what type of automated test do you think these test cases naturally map to? If we just blindly take test cases and automate them, what type of test would typically be created?

With generally accepted Agile documentation techniques, requirements are communicated to testers in higher level, user centric language (often, we even call them user stories). Think "Given-When-Then" stories and "As a… I want… so that…" acceptance criteria.



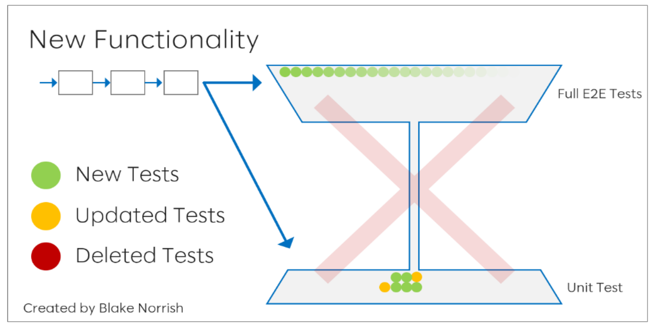
Story 1538 View Previous Orders

As a returning customer **I want** to see my previous orders **So that** I can quickly repurchase items	**Given** I am on the landing page **When** I click on 'previous orders' **Then** I see a list of all previous orders **Given** I am on the landing page **And** I have no previous orders **When** I click on 'previous orders' **Then** I see 'no previous orders'

A very hypothetical Agile user story… what type of tests would be created from this?

Even when stories are sliced horizontally and describe the behavior of a specific component (e.g. an API of a REST service) the requirements will be communicated in "user language" of that component. Thus, the test cases created from this documentation naturally map to the larger, more risky types of test automation.

This is the root problem of the automation factory approach — automating test cases inevitably over-emphasizes large, slow, and expensive tests, because test cases are naturally written in the language of the manual tester. They map to the exact type of test that our value-over-time analysis told us to avoid!



All new functionality is tested with dev unit tests and new E2E tests, leading to a top-heavy and hard to maintain test suite!

A second and sometimes just as powerful driver pushing automators to incorrectly prefer large E2E tests over smaller tests is that it is psychologically reassuring for non-technical people (and, some technical people) to know that test cases have been automated. For example, business leaders can understand test cases because they describe application functionality in language they are familiar with, and knowing that these cases are automatically being checked, reassures them that they won't be getting 2am calls from angry customers. They get far less reassurance knowing that the dev team has 90%+ unit coverage.

Thus, some people will push percent-of-tests-automated and number-of-tests-automated metrics because it makes them feel more comfortable with the state of testing, not because it is actually a more effective or efficient method for automatically checking system behavior. Pushing every test case to be automated might make you feel safe, but will not create a healthy automation suite.

Symptoms of the Automation Factory

Using the automation factory approach of test-cases-in, automated-test-cases-out automation approach tends to lead to some very problematic but unfortunately common symptoms:

- Test suites that take many hours to execute, or that can only be run over night
- Automation teams whose sole purpose is to investigate then triage the failures of the previous execution—which consumes most of their time
- Test failures where the accepted mitigation is simply to rerun the test until it works
- The removal of the suite from the CI/CD pipeline, or demoting it to a non-blocking step
- Test suites where developers avoid or outright refuse to run them because they don't trust the results
- Suites with thousands of tests, spread across hundreds of folders (or even different repos!), with duplicated tests, commented tests, and tests nobody knows what they do or how they got there
- Herculean efforts by automation engineers to manage the bloated suite of tests, or to hide the complexity behind a layer of gherkin (eg: Cucumber, etc.)

All of these symptoms are indicative of a test suite that is not providing value to the team that owns them, which is unfortunately common within development organizations. The suffering teams commendably prioritized test automation within their development process, but approached it with a naïve automation factory mentality.

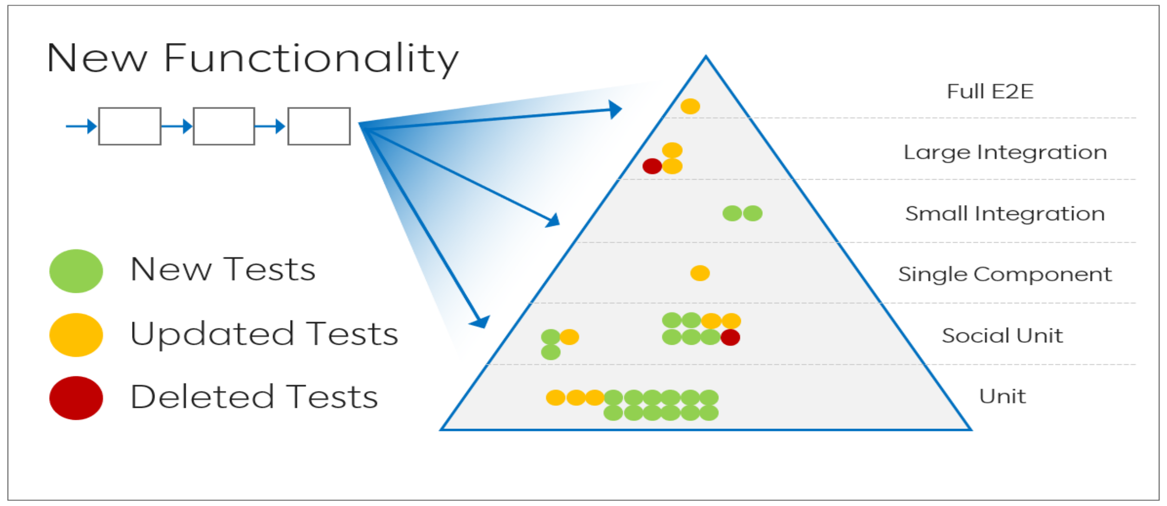
Healthy Automation

Ok, so how should you approach test automation to avoid the bloated suite?

The need for automation covering new functionality should be evaluated holistically by looking at automation options across every type of test. You absolutely do not want to assume that new functionality necessarily needs a new highest-level automated E2E test. Instead, evaluate how the functionality could most effectively and efficiently be covered over the full set of all test types.

You don't want to automate test cases, you want to automate functionality. Functionality can be partially described by test cases, but automating every discrete test case into its own automated test, at the level the manual tester would perform it, will never be effective or efficient.

In fact, the most effective way to test new functionality might simply be to update an existing automated test, move the test into a more appropriate test type, or even create an entirely new test type. Don't forget that as system functionality changes, you should be looking to delete tests as much as you are looking to add them!



Don't assume new functionality needs new E2E Tests, consider all type of automation!

While the type of test you need will be highly dependent on your system architecture, acceptable risk profile, existing tooling, etc. A generally healthy approach to automation changes would look something like:

1. Add new tests at the lowest level possible.
2. Update existing tests to cover new functionality.
3. Remove any now-obsolete or redundant tests, or combine tests.
4. Test the general case at a high level, then move permutations of that test into smaller, lower types of test.
5. Add new, high level E2E tests only if absolutely necessary.
6. Introduce a new type of automated test if this functionality cannot be covered by any existing type of test.
7. Modify the system architecture to enable a new type of automated test.
8. Continually and critically evaluate the health of the full suite of all test types with the full development team.
9. Point number 7 above deserves special attention as it is a critical difference between healthy automation approaches and automation factories.

As development teams, we must stop thinking about test automation as something that is applied to software only after that software has been built. Instead, automation should be considered a critical part of the software development process itself. Automation must be grown with software and the requirement for automatability should drive system design and architecture just as much as any other design requirement. Approaching automation in this way will enable smaller, more economical types of test automation unavailable to systems built without consideration for automatability. Healthy architecture is designed to be tested, and the role of the automator is just as much to inform system designers of these requirements as it to write automation after the system has been built.

Understanding that all automation has a cost and those costs create risk, that functionality should be tested with many different types of automated tests, that the challenge of test automation is leveraging these types of tests appropriately to holistically create the most effective and efficient suite of automation, and realizing that automatability is as important in system design as any other requirement — this is the healthy approach to test automation.

Creating automation factories and blindly automating all test cases as new top-level automated tests is not.

References:

The internet has a ton of great material on healthy test distribution, types of tests, test investment, and many of the other subjects discussed in this article. Unfortunately, it's buried in a lot of fluff, uninformed speculation, and marketing material. Here are some of my favorite resources on these topics:

- The Practical Test Pyramid, Ham Vocker
- Testing Strategies for a Microservice Architecture, Toby Clemson
- The Diverse and Fantastical Shapes of Testing, Martin Fowler
- Write Tests. Not too Many. Mostly Integration, Kent C. Dodds
- The Testing Trophy and Test Classifications, Kent C. Dodds
- Testing Pyramid Ice-Cream Cones, Alister Scott
- Round Earth Test Strategy, James Bach
- Just Say No to more End-to-End Tests, Mike Wacker
- Testing vs Checking, Michael Bolton and James Bach
- The Regression Death Spiral, Blake Norrish (yes, me)
- Test Cases are not Testing, James Bach and Aaron Hodder
- The Oracle Problem in Software Testing, A survey IEEE TRANSACTIONS ON SOFTWARE ENGINEERING

Tea and Testing with Jerry Weinberg

JERRY WEINBERG
October 27, 1933 – August 7, 2018
–
Gerald Marvin (Jerry) Weinberg was an American computer scientist, author and teacher of the psychology and anthropology of computer software development.
For more than 50 years, he worked on transforming software organizations. He is author or co-author of many articles and books, including The Psychology of Computer Programming.

His books cover all phases of the software life-cycle. They include Exploring Requirements, Rethinking Systems Analysis and Design, The Handbook of Walkthroughs, Design.
In 1993 he was the Winner of the J.-D. Warnier Prize for Excellence in Information Sciences, the 2000 Winner of The Stevens Award for Contributions to Software Engineering, and the 2010 SoftwareTest Professionals first annual Luminary Award.

For over eight years, Jerry authored a dedicated column in Tea-time with Testers under the name "Tea and Testing with Jerry Weinberg". As a tribute to Jerry's and to benefit next generation of testers with his work, we are re-starting his column from this issue onwards.

To know more about Jerry and his work, please visit his official website http://geraldmweinberg.com/

Software Subcultures - Part 1

"I have had discussion with executives in hundreds of different businesses and industries. Regardless of the nation, product, service, or group I am never disappointed. Someone always says: 'You have to recognize that our business is different.' Because they usually see only their business, they never realize how alike businesses are. Certainly the technology and the methods of distribution can be very different. But the people involved—their motivations and reactions—are the same." - Philip B. Crosby

What Crosby says about business in general is certainly true for software businesses. In this chapter, we'll introduce the major groupings of software patterns, or sub-cultures, and relate them to Crosby's work that he summarized in his "Quality Management Maturity Grid."

Applying Crosby's Ideas to Software

Readers who have read "Quality Is Free" will notice how consonant our views of software quality are to Crosby's views of quality in general. In particular, they will notice that we share the view that the critical factor is always "the people involved—their motivations and reactions." Even so, few people have had much success in directly applying Crosby's approach to software engineering. That's because, as we've said,

1. No two organizations are exactly alike.

2. No two organizations are entirely different.

We have changed Crosby's approach to account for the differences, so we need to explain several areas in which our approach to software quality differs from Crosby's.

> Quality is the ability to consistently get what people need. That means producing what people will value and not producing what people won't value.

Conformance to requirements is not enough

Crosby is very clear in defining quality as "conformance to requirements."

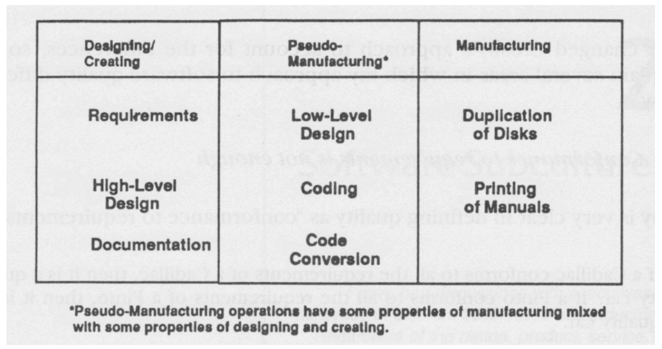
"If a Cadillac conforms to all the requirements of a Cadillac, then it is a quality car.

If a Pinto conforms to all the requirements of a Pinto, then it is a quality car."

That's an excellent definition as long as the requirements are correct. I'm not an expert in manufacturing, so I can't say how frequently manufacturing requirements are clear and correct.

I am an expert in software engineering, however, and I can definitely assert that software requirements are seldom even close to being correct. If the customer wanted a Pinto and you built a car that conformed to all the requirements of a Cadillac, that is not a quality car.

Many writers on software quality have missed the point that software development is not a manufacturing operation. It does contain a manufacturing operation—the duplication of software once developed. Indeed, some of my clients have successfully applied Crosby's definitions and approach to making accurate copies of completed software. Software duplication, however is generally not one of the most difficult parts of software development (Figure 2-1).



Designing/ Creating	Pseudo- Manufacturing*	Manufacturing
Requirements	Low-Level Design	Duplication of Disks
High-Level Design	Coding	Printing of Manuals
Documentation	Code Conversion	

*Pseudo-Manufacturing operations have some properties of manufacturing mixed with some properties of designing and creating.

Figure 2-1.

Some of the processes in software development are manufacturing operations and some resemble manufacturing in a few of their aspects. These can definitely apply Crosby's approach to achieve high quality.

In software development, therefore, we've had to generalize the definition of quality:

Quality is value to some person(s).

Requirements are not an end in themselves, but a means to an end—the end of providing value to some person(s). If requirements correctly identify the important people and capture their true values, this definition reduces precisely to Crosby's conformance to requirements. In software work, however, we cannot assume this ideal situation, so much of the development process is concerned with more closely approaching the "true" requirements. Therefore, much of what we need to understand about quality software management concerns this parallel development of requirements and software.

"Zero Defects" is not realistic in most projects

Because software development is only partly a manufacturing process, Crosby's goal of "Zero Defects" is not realistic. It is realistic for the manufacturing parts of the process, such as code duplication and probably coding itself (once the design is accepted as a true representation of the true requirements). And perhaps, in ten or twenty years, it will be realistic for the design process itself, at least the low-level design (Figure 2-1).

However, in 35 years of software building and consulting, I've never seen anything approaching "Zero Defects" in requirements work. If you examine those software projects that claim to be "Zero Defects," you will find that they always start with an accepted requirements document, as in,

- Conversion of a program from one language to another, where duplicating the behavior of the original program is taken as the absolute requirement. There are now companies that can consistently do such conversions on fixed schedules with fixed prices—and "Zero Defects" guaranteed.
- Creating a program for a new environment, using a standard requirement , as in the creation of a new COBOL compiler.

Thus, for the foreseeable future, most of us will have to manage software development in a "dirty" environment, where requirements cannot be assumed correct. To ignore this reality would be to play the ostrich, not the quality software manager.

There is an "economics of quality"

Crosby says, "The third erroneous assumption is that there is an 'economics' of quality. ...The second (most often offered excuse managers offer for not doing anything) is that the economics of quality won't allow them to do anything. What they mean is that they can't afford to make it that good....If they want to make certain that they are using the least expensive process that will still do the job, they should get deep into process certification and product qualification."

Again, this assumes that there is a correct set of requirement to start the process. If the requirements are correct, then it is not the development manager's job to decide what is "gold plating" and what is essential. The requirements answer all such questions once and for all. If there is only one right way, there cannot be any question of the "economics of quality." As Crosby correctly says,

"It is always cheaper to do things right the first time."

However, when the customers' values are not known, and even worse when the customers are not known, then we don't know what the "things" are. We may produce things right, but discover that they are not the right things. That's why the requirements process can produce or destroy value, and that's why there's an economics of quality, in any software project that includes a requirements process.

This "economics of requirements quality" certainly argues for getting the requirements right in the first place. If you can do it, then by all means take that approach. Where you cannot, however, the politics/emotions of negotiating value (quality) will permeate your project—and make it much harder.

Any pattern can be a success

In the examples of the previous chapter, we saw that even errors in conformance to formal requirements don't necessarily destroy the value of a software product, and that trying to meet every last requirement can result in destroying value for a subset of the customers. That's why the battle cry of so many software development managers is

Don't touch the program! or even more conservative,

Don't touch the (software development) process!

Although this attitude is often ridiculed, it makes sense economically. If the way you now produce and maintain software is wholly satisfactory, don't work on changing it; work on maintaining it. If your customers are happy, it would be foolish to change.

As we'll see, collapse (of a program or a process) is an ever-present possibility for most software managers. If your customers are mildly unhappy, then you're probably in the right pattern, but not doing it as well as you could. Don't change your basic pattern, but improve it by small, safe changes that don't risk collapse.

But if you're currently in the wrong pattern, then trying to improve it by small changes is like creating ever more detailed maps for the wrong trip. If you're supposed to go from Miami to Cleveland, then detailed maps of the Los Angeles metropolitan area are not only useless, they are distracting.

If your customers are unhappy, it will be fatal not to change. If you're not in the appropriate pattern, then choose the pattern that will give you the quality/cost you need and work within that pattern to do it well.

Quality is the ability to consistently get what people need. That means producing what people will value and not producing what people won't value. Don't use a sledge- hammer to crack a peanut. Don't use a nut-cracker to break up a wall. Choose the pattern that will give you the quality/cost you need and work within that pattern to do it consistently.

Working consistently is the essence of a pattern, or sub-culture. Working consistently to give value to your customers is the essence -of success. Therefore, any subculture can be a success.

"Maturity" is not the right word

It's very tempting, when writing about cultures, to slip into a judgmental mode. For instance, it's hard for some people to believe that any software subculture can be a success. Like the pigs in Orwell's Animal Farm, they accept the words that say, "All animals are created equal," then add, "...but some are more equal than others." "Any software culture can be successful," they agree, "but some are more successful than others."

Most often, this judgment slips in covertly. Crosby, for example, describes five different patterns of quality management in his "Quality Management Maturity Grid." The Grid is a strikingly useful tool, but a better name would have been, simply, "Quality Management Grid." The word "maturity" is a judgment, not a fact, but an interpretation of facts. At the very least, it doesn't fit the facts. Maturing normally goes in one direction, but Crosby gives several examples of organizations "falling back, as in this quote:

"We were Enlightened (one of the "maturity" stages) for a couple of years, then we got a new general manager who thinks quality is expensive. We'll have to drop back a stage or two until he gets enlightened."

In everyday language, "mature" means "having attained the normal peak of natural growth and development." There's nothing particularly "natural" in the progression through Crosby's stages. Indeed, Crosby is at great pains to emphasize the vast amounts of work involved to change from one stage to another.

Moreover, I have observed many software organizations that have attained "the normal peak," in the sense that they are going to stay right where they are unless something abnormal happens. They are good enough, and investing in attaining another pattern would serve no organizational purpose. As we've seen, cultural patterns are not more or less mature, they are just more or less fitting. Of course, some people have an emotional need for "perfection", and they will impose this emotional need on everything they do. Their comparisons have nothing to do with the organization's problems, but with their own:

The quest for unjustified perfection is not mature, but infantile.

Hitler was quite clear on which was the "master race." His "Aryan" race was supposed to represent the mature end product of all human history, and that allowed Hitler to justify atrocities on "less mature" cultures such as Gypsies, Catholics, Jews, Poles, Czechs, and anyone else who got in their way. Many would-be reformers of software engineering start their work by requiring their "targets" to confess to their previous inferiority. These "little Hitlers" have not been very successful.

Very few healthy people will make such a confession voluntarily, and even concentrations camps didn't cause many people to change their minds. This is not "just a matter of words." Words are essential to any change project, because they give us models of the world as it was, and as we hope it to be. So, if your goal is changing an organization, start by dropping the comparisons, such as implied in the loaded term "maturity."

Six Software Sub-Cultural Patterns

To my knowledge, Crosby was the first to have the idea of levels of process maturity. He noticed that the (mostly) manufacturing organizations with which he worked could be studied according to the quality of their production. If he knew the quality of their product, Crosby could make predictions about what practices, attitudes, and understanding he would find inside the organization.

Crosby's observation was something we organization consultants use all the time, an application of "Boulding's Backward Basis", which says,

Things are the way they are because they got that way.

In other words, you can study products to learn about the processes that produced them, in much the same way that archaeologists study levels of technology from the remains they dig up from ruins. Like the archaeologists, Crosby discovered that the various processes that make up a technology don't merely occur in random combinations, but in coherent patterns. Crosby named his five patterns:

1. Uncertainty
2. Awakening
3. Enlightenment
4. Wisdom
5. Certainty

based largely on the management attitudes to be found in each.

In their article, "A Programming Process Study," Radice, et al. adapted Crosby's "stratification by quality" scheme to software development. In his book, Managing the Software Process , Watts Humphrey picked up their work and identified five levels of "process maturity" through which a software development organization might grow.

These patterns were called:

1. Initial
2. Repeatable
3. Defined
4. Managed
5. Optimized

These names were more related to the types of processes found in each pattern, rather than to the attitudes of management.

Other observers quickly noted the usefulness of Humphrey's maturity levels. Bill Curtis, of MCC, for example, noticed that a parallel classification could be made simply on the basis of the way people were treated within the organization. He proposed a "software human resource maturity model" with five levels.

1. Herded
2. Managed
3. Tailored
4. Institutionalized
5. Optimized

Our own work with organizations is guided by the anthropological model of "participant observation," so we tend to observe what's happening at the bottom levels, not just what management is doing and saying. We particularly look for the degree of congruence between what is said and what is done in different parts of the organization. Classifying organizations by their degree of congruence, we can match them to the other systems of patterns as follows,

1. **Oblivious**: "We don't even know that we're performing a process."
2. **Variable**: "We do whatever we feel like at the moment."
3. **Routine**: "We follow our routines (except when we panic)."
4. **Steering**: "We choose among our routines by the results they produce."
5. **Anticipating**: "We establish routines based on our past experience of them."
6. **Congruent**: "Everyone is involved in improving everything all the time."

This is the classification we'll use throughout this article to describe organizations.

To be continued...

PART 2 IN NEXT ISSUE

Getting noticed is easy.
Getting noticed by the right people is hard.
Advertise with us.
Connect with the audience that matter.
Contact us: sales@teatimewithtesters.com

INTERVIEW

Failure Resume? What are those? What makes one an effective test leader? How has testing changed in recent times? Find it all in this exclusive interview with Anuj Magazine

- INTERVIEWED BY LALIT BHAMARE

Anuj, what a pleasure talking to you today after quite some years we discussed testing last time. Looking at where you are today, a lot seems to have changed. Would you like to briefly tell us what are you up to lately?

On the work front, I currently work with Walmart Global Tech India and lead Strategic Technology Programs. This role is primarily in the Strategy domain, quite an open-ended job description which is (currently) focused on building world class tech culture in the organization. In the last 8-10 years, I have consciously worked to pivot my career and been fortunate to have taken up the roles that I never did before. It is discomforting to move away from your expertise every 2-3 years but I like it that way. The best perk for me is that I am not in a rat race (of chasing next designation) anymore.

My career has been career-path agnostic in many ways as I have worked to drift away from a well-set path defined by someone else. Growth is usually discovered at a zone where the comfort zone ends.

Beyond work, much of my time is spent with family, which is the first dimension of my life (work being second). My hobbies and interests form what I call the third dimension of my life as these have helped positively augment and influence the other two dimensions- Work and Family life. My intent is to pick-up a new hobby every year and put in hard yards to master it. I have pretty much managed to do this in the last 14-15 years.

While we are discussing your career and accomplishments, please tell us how “testing” helped you in your journey.

To put things in context, so far, I have worked in the following domains- Testing, Development, Globalization and Security Engineering, Technical Operations, Product Management, Strategy.

My approach in entering and mastering a new domain is simple- start with a mastering mindset, follow-up with building relationships, then focus on skills and mastery. (pretty much in that order).

Software Testing was my first job and I did it for sizable time in my early/mid career so it has a great influence in my career, going by sheer volume of time spent. I still try and test and explore in my spare time. To me, the key elements of testing mindset includes- caring about bringing the best for the product (and for the company), striving to bring clarity in decision making, seeking the truth, always striving to shorten the feedback loop.

Let me give you an example of how testing mindset helped me in my recent role. We were engaged in formalizing the strategy for a crucial area. After having put all the details- including vision, mission, and path to achieving the mission, i felt something was missing. The proposal that i had covered mostly the linear scenarios i.e. if all the said dependencies were met, then we will achieve what we set out to do. Building an impeccable strategy is an iterative process. In the next iteration, inspired by the inverse thinking mental model, I introduced the pre-mortem process. In the pre-mortem process, we collectively worked to identify the failure points i.e. covered all the angles why our strategy would fail. This exercise helped us pre-empt failures even before they occurred. Inversion principle, simply put is, Avoiding stupidity works better (in most situations) than trying to be brilliant.

My application of Inversion mindset is tied directly to application of software testing mindset, which I learned to practice years ago.

ANUJ MAGAZINE
–

Anuj Magazine currently works at Walmart Global Tech India and leads Strategic Technology Programs. Prior to this, he worked at Citrix and handled leadership roles in a myriad of functions like Product Management, Technical Operations, Engineering, Security during his tenures. He has also played different roles in technology companies like McAfee and Quark.

Anuj has 16 patent filings.

If I am not wrong, you have also worked on the product side of building software systems. Would you like to tell us how best testers in a team can help product people, especially in the era of DevOps?

Product profile is unique in a sense that it has intersection with almost all the functions in the organizations. For testing specifically, i have seen the following ways the value was added with testing-product collaboration:

1. Present easy-to-consume information about product health status. The keyword here is 'easy to consume'. The truth about product health shouldn't be hidden beneath multiple layers. And testers should be upfront and candid about sharing the bad news, if need be.

2. Testing functions can partner effectively in the value discovery phase. A/B testing is usually done by product teams. Testers can scale-up to understand the product discovery phase requirements and set them up to partner with the product teams.

3. Testing can partner with product team to help assess the field feedback about a feature or a product, and play a role in shortening the feedback loops- swiftier delivery of fixes and experiences to the customers.

4. Testing can help facilitate a large-scale dogfooding programs to test the features in-house before the product is launched fully.

These are just a few examples but the core idea is- If you do not limit your thinking, testing can collaborate with almost any and every function in the organization and add never-thought-of value. Having an open mind is such an underrated growth hack.

I guess I would never stop mentioning your fantastic article that we had published in one of our previous issues, Software Testing and the art of staying Present. Does staying in present still matter for testers when the speed of deliveries and deployments are always pushing them to stay ahead of the game?

Thank you for the compliment and I am glad that you still remember that article. I believe that staying in the present is more important now than it has ever been.

We live in a consumption economy. The whole digital revolution seems to be built on the premise to offer the contents (in the form of news, updates from friends, images, videos etc.) to us as effortlessly as possible. We now have smartphones that are 24x7 content broadcasting machines. As a result of this, human beings are always in content consumption mode. While access to information is good in one way (it has made us more aware) but largely it has also robbed conciseness from the day-to-day communication. The emails tend now to be longer, verbal updates muddled up and our white-boards more busier than they have ever been.

One of the pieces of writing that recently inspired me is a blog titled- Create less, Consume more by Tanmay Vora. Sharing couple of pieces of advice from this blog:

*Consume mindfully by having right set of filters that help you decide if something will *really* add value and increase your ability to create. When you consume mindfully, less is actually more.*

I believe in practicing the fine art of subtraction – we don't need more and more. We need less that is more- useful or helpful or enriching. Being in the present expands our ability to create more and consume mindfully.

You did a very impressive talk in the FailQonf. It was about the Failure CV. Please tell our readers more about it and why it is important?

Our CV or resume is such a fascinating document. All that we do in that document is project how awesome we are, how well we have done things, how fast we have progressed, how many things we have accomplished but we rarely stop and ask ourselves how did we even get to that point ? What are the things that we did that made us so good ? How did we reach where we have?

We rarely talk about failures in our resumes. Name three accomplishments you're proud of. I bet that didn't take you very long. Now name three failures that you're proud of. You had to think harder, right?

We live in an age that's supposed to celebrate failure. Fail early, fail fast, treat failure as an asset. But how often do you actually brag about your failures?

Including a Failure section in your resume helps fill that gap. But you might wonder why it is essential to talk about failures when we can get away with just talking about our successes in the interview process. I have used Failure resumes for the last 7-8 years. And in this period, I have worked in 3-4 different functions. With this experience, i have observed the following benefits of being upfront about failures in the resume:

1. It helped to build trust. Being seen as someone who is comfortable with talking about failures makes you appear more human. It helped traverse the challenge. of earning trust in less time especially when you are in interview situations. The more senior a role you are chasing, the more important role trust plays.

2. It helped to conquer the fear of failure.

3. It helped build Self-awareness . I've learned about myself from nearly all my failures, additionally, I've also learned not to take success for granted.

4. In one of the organizations I worked for, 'Courage' was one of the 4 core values. With my failures listed in my resume, I presented the proof of courage without me having to say a single word in my defense.

My general experience has been that engineering managers and leaders in organizations who get to decide about testing usually do not have formal education or hands-on experience with testing as such. This usually makes it hard for testers to feel understood and even communicating the value of meaningful testing becomes difficult. What would be your suggestion to change this scenario for good? Who needs to change and how?

There are 2 lenses i choose to look at the situation you describe-

1. External forces lens

Speaking not only of Software testing, but in general for all the functions, our industry has changed beyond recognition in the last 2 decades. Not just change, but the rate of change is unprecedented in these times. These are some of the trends that i have seen that continually tend to exert pressure on the software testing profession-

A. SMAC revolution- Social Mobile Analytics and Cloud trends has not just had impact on our lives but also on the way Software engineering is done. As an example, companies live Google and Amazon (and many others) performs 1000s of releases every day. The shrinking of release cycles has caused the role of testing to be reinvented.

B. AI/Data Science revolution- The advent AI continues to be a threat to all the routine, monotonous aspects of every job (not just some parts of Software testing) on earth.

C. Small teams- WhatsApp Android team that reached billion+ downloads for a large part of it's existence was handled by just 4.5 engineers (0.5 being a person who was engaged in configuration management). Businesses realize that with evolved release processes/automation, more value can be derived from less investments.

D. Delayering- Many organizations have cut additional layers that were causing inefficiency in communication and delivering value. Again not just testing but all functions have seen varying impact of delayering.

Likewise, there are many emerging trends (I haven't even called out Covid specific shifts but you get the drift here) that are putting a lot of pressure on the way Software testing is done. These trends have caused the value perception from testing being changed constantly. Leaders should be situationally aware of these trends and represent the value from testing appropriately. This is what brings me to the next lens.

2. Value lens:

Again, this is a generic observation but more often I have seen people not communicating the value that testing brings to the decision makers (who may not be from the testing domain). It is more a communication problem than a value problem. Leaders should increasingly focus on not just producing value but also how to package and communicate the value. One can make a great product but don't do enough hustling, then it won't sell. It is a high time we realize that producing value and communicating about the value (and convincing others) need different skills and application.

Looking at your passion for sketch-noting, it is evident that you have a great creative talent. How does that creativity help you in doing your work?

Visual thinking (that involves communicating via visual medium like images, videos etc.) is increasingly becoming important in today's times. Sketchnoting is an art form that helps to condense and communicate the relevant ideas. I honestly think i don't have innate talent in Sketchnote but it is more of an acquired skill (i didn't know how to draw 2-3 years back), which i got hooked on and working hard to hone. The only talent I have is being ruthlessly consistent and staying relentlessly focused. I see the following benefits:

Visual thinking helps you be more aware, be present in the moment:

Sketchnoting requires you to be deeply observant and have extended levels of self-awareness to summarize the ideas. really helps to extract more life out of each moment.

Visual thinking brings in brevity in communication:

Sketchnotes has really helped me balance the continuum of creation and consumption. Sketchnotes offer a powerful medium that lets you do a concise representation of a book or a large number of words in just one page. It really helps to separate signal from the noise. In short, it improves brevity in communication.

Being more intentional about listening:

Visual thinking has made me more intentional about listening. While the left side of brain is always busy understanding the content, segregating into different knowledge buckets, the right side of brain gets actively engaged in drawing pictorials metaphors of all the gathered knowledge.

Retaining more info:

The fact that you can draw the summary of a session or an article or a book or literally anything you like, helps to retain more info in your mind.

I loved this quote from David Heinemeier:

'I've realized that the hard part about most books is not reading them but recalling their knowledge or insight when you need it the most.'

Sketchnoting helps you achieve exactly that. The simple act of putting pen to paper helps you remember more but the fact that you can see the sketchnotes anytime you like (they are all over my work area) means that you always tend to actively or passively glance through this.This act helps in ensuring that you build strong neural connections in the brain.

”

Testers need high agency in high proportions to navigate through various challenges- from proving your existence to delivering value.

Building Networks:

Finally, one of the biggest blessings I got from my journey of visual thinking is that it helped me connect with various accomplished people, most of whom helped me make a better person and professional.

I remember I read this quote that stayed with me (I think from the book- 'Show your Work'):

Networking is less about knowing people and more about putting your best work out in the open. that attracts the best people out there.

You read a lot and write often. What books would you recommend testers that personally helped you?

Books have had a disproportionate impact on my life so I will certainly have recommendations. But since I read a lot, my list of recommendations also keeps evolving :-). Keeping testers in mind, here is what i would recommend:

The Third Door: The Wild Quest to Uncover How the World's Most Successful People Launched Their Careers by Alex Banayan

This book on the surface has nothing to do with testing, very less with technology yet i feature this high on my list. Reason- it teaches an important skill that I rate highly among professionals- High Agency. Let me explain what it is:

High Agency is about- "When you're told that something is impossible, is that the end of the conversation, or does that start a second dialogue in your mind, how to get around whoever it is that's just told you that you can't do something?" (Eric Weinstein's quote:)

High Agency is about finding a way to get what you want, without waiting for conditions to be perfect or otherwise blaming the circumstances. High Agency People either push through in the face of adverse conditions or manage to reverse the adverse conditions to achieve goals. (Shreyas Doshi's quote)

Testers need high agency in high proportions to navigate through various challenges- from proving your existence to delivering value.

Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives by Nick Rozanski, Eoin Woods

I recommend this book for a simple reason- I have seen a very few testers having a say in software architectures. Not that the tester's abilities are lesser than anyone else but it's more about orientation and developing interests. The best testers I have seen can present product architectures like seasoned architects would and add value in creating robust systems in a proactive way. This book can help you understand an architect's mindset.

Working in Public: The Making and Maintenance of Open Source Software by Nadia Eghbal

I recommend this book because I have seen a lesser number of Software testers participate in creating Open Source software. It's a fascinating read on the history, present, and near future of open source software development. I foresee a world where more and more testers participate in the Open Source community, holistically imbibe the Open Source ethos and help build better, cleaner software.

If there is one thing you would like to stop tester from becoming, what would that one thing be?

I read Subroto Bagchi's book- "The Professional" many years back and it had a profound impact on me. Subroto while describing the word "Professional" says- ""A professional who sees his work primarily as a means of earning money, runs out of meaning very soon.Being a true professional is nothing short of a religion and the capacity to serve is indeed a blessing in life."

He further goes on to say that there are 2 qualities that separate a professional from someone who is just professionally qualified-

Ability to work unsupervised.

Ability to certify the completion of one's work.

If there is one thing that I would like to stop testers from becoming, it is becoming unprofessional or non-professionals.

When we embrace a particular field as our chosen career, our responsibilities do not start and end at mastering the skills needed to execute or exceed the job expectation but it in reality goes much beyond. With extraordinary time and focus spent on building skills, we sometimes tend to ignore a larger view.

To make my point further, I have listed a handful of situations that we might face in our professional lives and followed up these situations with a question-

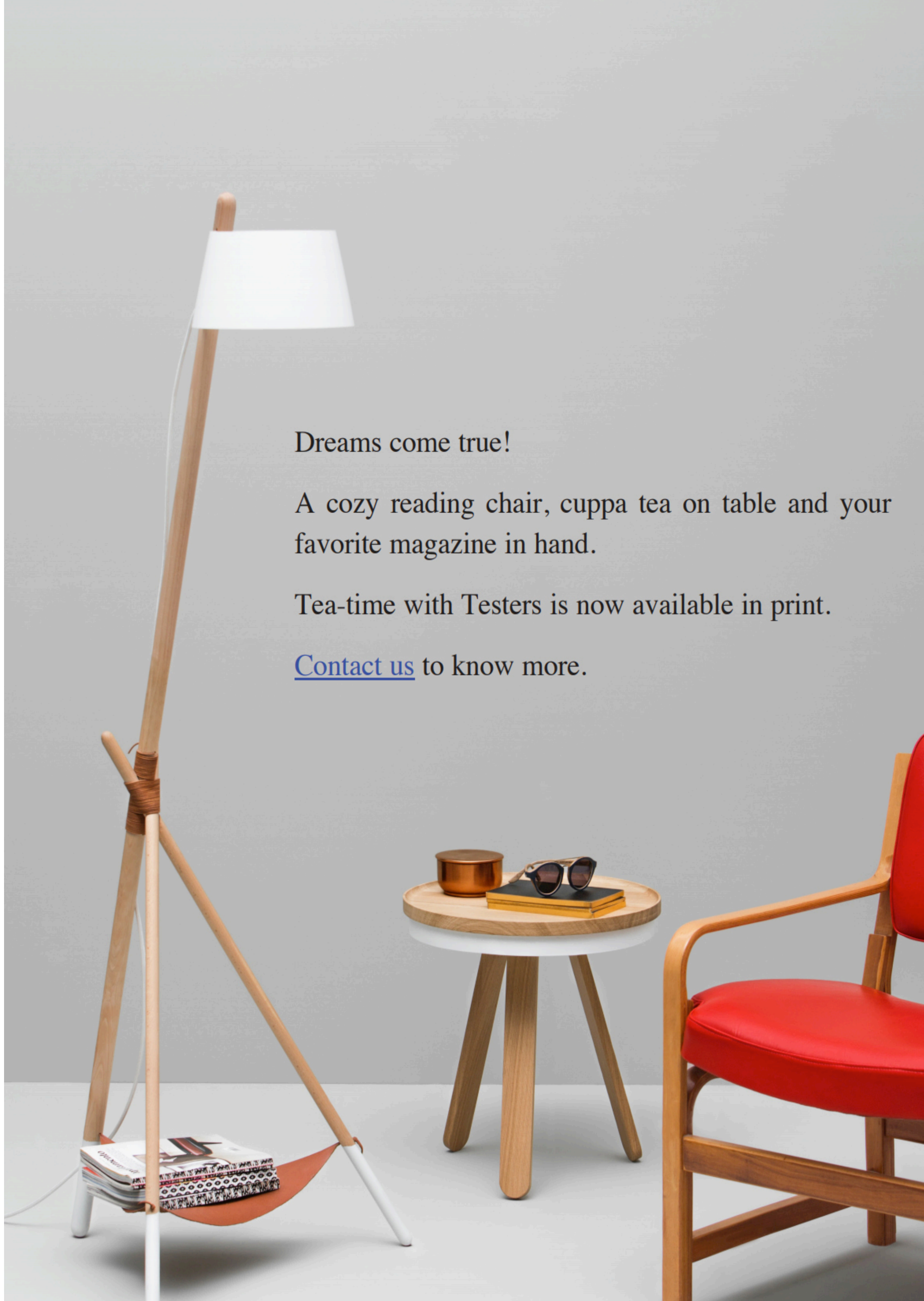
- A person finds a High severity, rarely reproducible defect in the Software component he/she was handling on a day before release. Should he/she go and inform the Management (despite fearing his lack of performance impressions) or should he remain quiet and not report the bug (as anyway it is rarely reproducible and will be rarely noticed) ?

- A person is knee deep in a technical problem, whose solution is likely to be available with another teammate. He/She does not quite reach the other guy for help just to serve his/her ego. Is it ok to let professional ego slow the pace of a project ?

- A person meets another colleague on a pathway, they have a discussion and as a follow-up, this person promises to send some information to the colleague in 2 days time. A week goes by and the colleague doesn't get the required information. Is it ok to be casual about the commitments made to people who are not your bosses or Managers ?

- A person installs a Software tool and learns it's very basic functions. Next thing he includes the very mention of the tool in his resume as one the "skills" he has. Is he right in claiming expertise on this tool (which may even turn out to be the basis of him getting an interview call) ?

Being a professional is our foremost responsibility as a tester, or for any other role/job you choose to take.



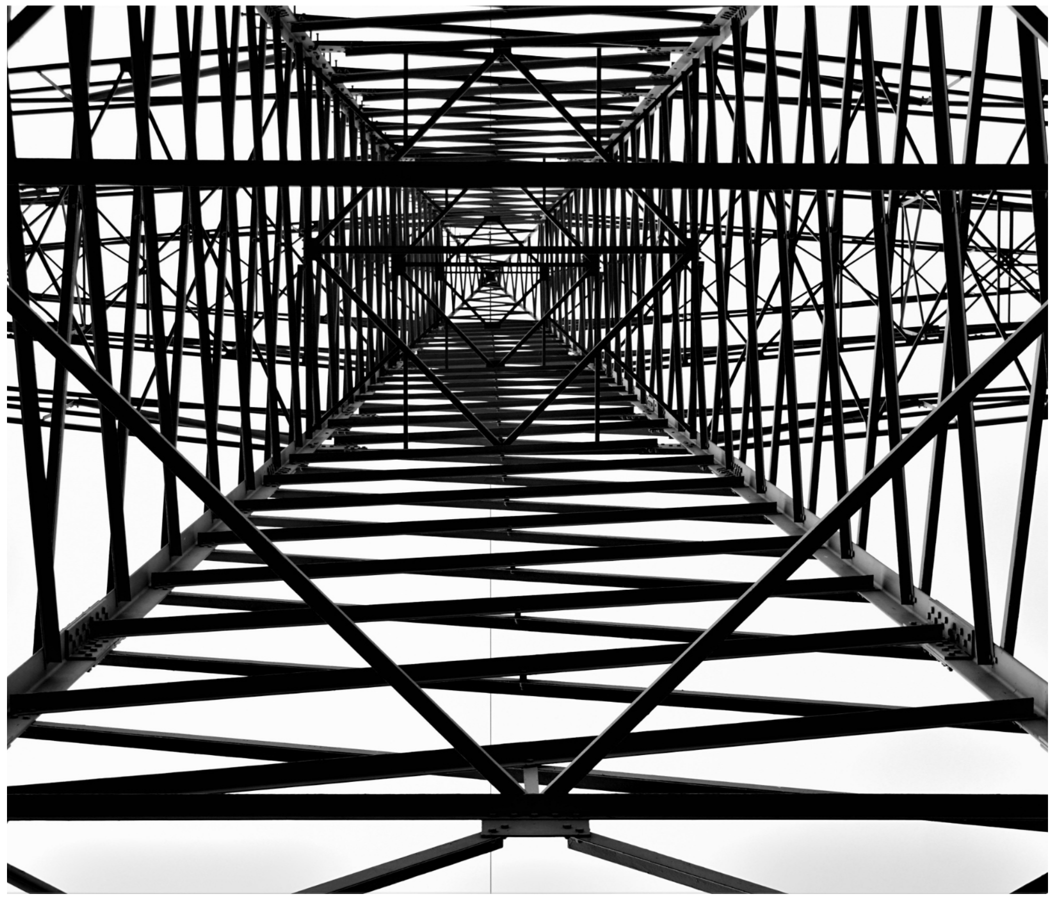
Dreams come true!

A cozy reading chair, cuppa tea on table and your favorite magazine in hand.

Tea-time with Testers is now available in print.

Contact us to know more.

~Processes



GUIDING PRINCIPLES FOR CHOOSING A TEST FRAMEWORK

"No one understands my passion for this vain test library, but I shall make it default for everyone"

– A vernacular Software Test Engineer

The problem

Test tool (or framework) comparisons, and the actual choice of a tool, can be a "painful" and overblown exercise in modern tech organizations:

- It's prone to a lot of lengthy and tiresome risk discussions between "hyper-focused" advocates of different tools that haven't used the tools of the competition;

- It's very easy in some cases for folks to fall for sales pitches from malicious test tool vendors;

- It's easy to fall for a "honeymoon"-like anti-pattern phase, where the initial impression/adoption fails to uncover a lot of deep rooted issues that come with prolonged tool usage.

- Lastly, tools are intimately dependent on their context, e.g. familiarity, team-level, org-level, and industry-level suggested ~~imposed~~ rules and standards, extensions & plugins, etc., meaning: the "one test tool/ framework/library to rule them all" (likely) does not exist.

This is oftentimes true regardless of the area of focus of the test tools, be it UI-facing, API-facing, Load/Performance-dedicated, ...

Proposed solution(s)

I'd like to share and suggest a handful of personal **guiding principles** that have helped me mitigate the non-useful noise and disruption that comes with a tool-choice exercise:

- **Abstract tool-specific complexity through accessible interfaces** (Anyone can run)

- **Containerize from the start** (Anyone can run from anywhere)

- **Abstract the "attack type"** (Anyone can run anything)

- **Expose meaningful test results** (Anyone can understand what failed)

- **Make debugging easy** (Anyone can tinker with failure)

The underlying principle is using **accessibility and maintainability** as a compass (and a shortcut) to help create some sort of safety-distance / critical perspective while comparing test-frameworks.

Let's look at each of the above points in detail.

" Whatever complexity we might code into our automated checks, we need to provide multiple accessible interfaces for folks to run the automated checks, abstracting any tool-specific complexity (and setup) at all its levels.

Anyone can run

One of the most overlooked aspects of any test tool, and specifically tooling that is adapted and then developed on top of inhouse tools is that it's not always straightforward for folks to make use of the actual tools.

If you have been working on the tech industry for a while, you probably have seen some of the worse symptoms of what I'm describing:

- The automated checks are only run by the same folks developing the checks from the UI of an IDE, e.g. something done by Jetbrains like IntelliJ, or done in any sort of the typical apps nowadays built with ElectronJS, like Insomnia, Paw, Postman, etc.

- The automated checks only run locally through a CLI or a set of shell commands, and are a mess to setup locally on someone else's machine aside from the machine of the engineers coding the checks;

- The automated checks are already at a stage where they run on a CI pipeline in our typical Jenkins/GitlabCI/Github Actions/… BUT, only the engineers that created those checks know about the CI jobs, and are the only ones checking the results of those jobs;

- …

The basis of the problem is always the same: only the test engineers developing the checks know how to run them and/or how to interpret the results, and there is a massive disconnect in value between the automation and everyone else in the team.

In my personal experience, to fight this problem we can resource to one principle:

Whatever complexity we might code into our automated checks, we need to provide multiple accessible interfaces for folks to run the automated checks, abstracting any tool-specific complexity (and setup) at all its levels.

What this means always depends on the context where we're working, but here's a few good signals that we should be tuning for:

- If non-technical folks are drawn to run the tool and can run it unsupervised and without fear of causing damage;

- If technical folks can run the tool in a way that suits their preferred development environment tempos;

- If the interface we designed is easy to remember;

-…

In the field the above can look like this: suppose we're working on a test tool called "testthis" where folks can mimic user flows pointed at the API level.

Technical folks can run the tool via CLI, which could look like:

testthis run --flow release_goat_for_trex --environment staging --project dinopark

And non-technical folks can run the same tool from points where they are used to work in, like in Slack or other chat-based software, e.g. using a slash command:.

/testthis run --flow release_goat_for_trex --environment staging --project dinopark

The trick is abstracting and reducing configuration, reducing the mental load of someone using the tool. There's more I could write about this principle alone, because it's tricky to design an interface that is simple but not over-simplifying. I'll leave it to another post for the time being.accessible

Anyone can run from anywhere

One of the coolest things that I've come to appreciate over this past year was having direct access to a friend who also happens to be a Docker Captain (yes, that's right Tom, I'm name dropping you on one of my posts).

The main teaching that we adopt when we're interacting everyday with someone that is a Docker (and containers) ace pilot is:

If we can break apart and contain a solution to a problem, e.g. a piece of software, the solution might not be perfect, but we're solving in one go a lot of other tiny issues for ourselves and others.

We go from the typical "Works on my machine" to being able to share and distribute the thing, contain it and pin its dependencies to a working state, and reuse it without worrying about internals or obscure steps of setup or host machine specifics. It now "works on my container", which is a slightly better predicament than having something just work on the programmer's host machine.

Sadly, what we'll find inside most organizations in the industry is that, aside from using existing containerized tools, very few test engineers tap into that power and start thinking about containerizing their own automated checks and their own in-house test tooling, and fail to extend tools that already allow for containerization. I believe this also aggravates the pickles that a lot of Test Engineers endure:

- "No one uses my test tools"
- "No one cares for the automated checks I've coded"
- "My test tools work fine, folks just need to follow these 10 steps on their machines to set it up, and between executions do these 15 steps to reset the test data"
- "I've worked on this piece of automation for months, and no developer or tester or non-technical person uses it"
- …

Nobody cares because the test engineer is not distributing the thing properly.

Distribution is not just sharing a link to a repository or a CI job. In order to fix these "dormant test tooling dilemmas, we need to keep in mind:

- Any piece of test tooling we are building needs to provide value from the start, not in "6 months";
- If there's no easy to follow README showing me how I can run the tool, I absolutely don't care for the tool;
- If I can't just spin up a container of the tool, not even technical folks will care for the tool.

And probably the key side-pieces that come as a byproduct of the above principles:

- We can provide value from the start if we can get it in the hand of folks that will use it,
- There's no better way of getting it to folks hands than containerizing it,

Containerizing takes you to the next level, because containers can "run anywhere" and be triggered to start "from anywhere",

"from anywhere" means we easily provide our tool through a chat bot, a spreadsheet, or any other tool that can underneath do any sort of API requests to run a container "somewhere".

When we do the exercise of putting our test tool or our handful of automated checks working in a way that they can be run "anywhere", more often than not it also forces us to think about the next problem: to "run anything" pointed everywhere.

Anyone can run anything

It's usually the case our test tools all try do the same: they try to follow a scripted path of the interaction of a user through a certain narrow perspective of a product.

If we wanted the test tool user to do this same interaction at a larger scale, like in a load test, it would be a good practice that they could easily just do that: indicate that they want to run the same thing they are running mimicking one user - but for hundreds, thousands, etc… of users.

Here's the part where most Test Engineers will spot a gotcha: folks dedicate too much time either:

- on tools that accomplish flows for a single user, where they add a lot of detailed assertions throughout those,
- or attacking the scale problem, focusing on load tests that are not deep in assertions.

But they almost never dedicate balanced time for both. This cannot be the case.

My proposed principle to try and do things right in this case is that we need to push for ways that we can dedicate enough meaningful time for both implementations. And this is only possible if from the start we try to provide those through the "same" interface:

testthis run --flow some_flow (… other arguments)

testthis run-load --flow some_flow (… other arguments) (load specific arguments, like number of virtual users, iterations)

testthis run-distributed-load --flow some_flow (… other arguments) (load specific arguments) (distributed arguments)

The end user should be able to run anything easily, they just need to focus on choosing the right "attack" type, and the "parent" test tool abstracts the underlying complexity, and acts as an alias of any other underlying tools.

Anyone can understand what failed

This goes back to something I had mentioned in this post.

As test engineers we tend make it so when a given automated check suite fails, we get notified with some bland message and a link to a CI job. Problems with this approach:

- The notifications end up being cryptic for the desired audience for those notifications (in theory the whole team),
- It's dumb to have to go through logs if what you want is just an **initial understanding** of the issue,
- The notifications don't provide any meaningful information, their intended premise is "folks should care about these", which curses the notifications to fall into oblivion, since they quickly get ignored.

This is not the way. Notifications for test tool failures should be as "delicious", enticing and meaningful as a typical predatory notification for a new social media post… without the shallowness and ad-revenue hungry demonic spirits that come with default social media notifications and clickbait.

What would this look like in theory? Well, it means we prioritize:

- Meaningful error messages and easy to access logs over bland failure messages;
- Direct-feedback loop over scripted test case loops through integrations.

And what does this look like in practice? Taking the example from my anti-patterns post, it's all about trying to reach a message that tells a story, like this:

SomeAutoChatbot says: The endpoint ABC in development environment 029 is failing with 502 Bad Gateway for the buy-an-action-man test scenario. Error trace-id is 053de188-7438-42b1. Link to the logs some kibana/cloudwatch link. Possible solution: restart the orders service here or contact @oncall-support-dev-env-team.

versus saying something bland, like this:

something is not working, please check my failed jenkins job and the ticket 1234 of the test case on JIRA

The point is: whatever you do, you optimize for the message itself, by being sure that anyone in your surrounding context, including yourself, can have a quick grasp and clear signals of why something failed, and you leave breadcrumbs for folks to investigate deeper if they are up for it.

Anyone can tinker with failure

How many times has a developer reached out to a test engineer and asked - how could I do this specific automated check or debug a failing check, only to be met with several flavours of the same response:

You could, but you can't

We tend to make our lives in a project harder by not looking at probably the most useful problem to look at after the problem of containerization:

How do we make it easy for anyone to debug a failure state of our test tool?

This principle depends heavily on the programming language, libraries and tools from which each of us build our own automated checks and test tooling, but its importance is what can make or break a test tool and even a test engineer.

Some folks are quick to write this off and will say: "Ah, if folks do this and that on the specific dev environment that I use, they can somewhat debug the test tool… problem solved"

Those folks fail to realize they are a part of the problem. This shouldn't be the case. There are a few steps I can suggest in this case:

- You should be able to provide multiple different ways that folks can both "breakpoint-resume" debug the test tool, as well as your typical "console log" approach;
- The tool you are building details somewhere in meaningful logs and other document formats the steps it took while trying to follow along a certain flow;
- Any debug approaches should be documented somewhere where it's easy to find the info;

- You should try each of the debug approaches you suggest for yourself;
- If any of the suggested debug approaches is hard to explain or convoluted and complex, scrap it for a simpler one.

Wrap-up: A caution word about tools that try to be human

Here's what most folks might not talk about when it comes to test tool choice: the "evil" of tools that try to be and do everything a human does.

By this I mean:

They suggest ~~impose~~ a human-based domain specific language (DSL)

There's two crucial points to keep in mind regarding this:

- It indirectly promotes software testing busywork, as in, "it is work that is done to show that work is being done"
- It impacts maintainability, since not only you have to maintain test code, you also need to maintain a regurgitated human translation of that code that no one will care for;
- ....

Just these points breed the equivalent of the flowers blossoming problem: weeds blossom due to (re)implementation freedom. Oftentimes you end up with an extra million ways of using the "do-it-all" library to solve a certain path within the same org, plus the added clutter of having libraries that do more than what you are trying to do in the context of a scripted test.

So, I'll wrap up this post with a word of caution:

There is such a thing as test tools/frameworks that try to do everything a human does so much so they become vain tools.

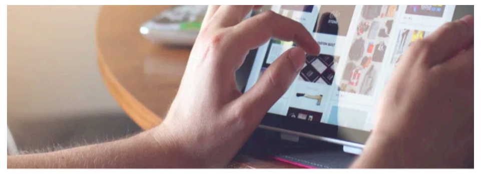
I recommend you give a read of Michael Bolton's experience reports of Katalon and mabl to get a feel of what this usually means from the lens of a hardcore software tester.

FILIPE FREIRE

Filipe is a portuguese test engineer, currently working with Kong Inc. In the past he's worked doing all sort of testing and test engineering efforts in different software projects.

Outside of work he often blogs about testing, videogames, and more at http://filfreire.com.

TEA-TIME WITH TESTERS
AN INTERNATIONAL JOURNAL FOR NEXT GENERATION TESTERS
Learn new things this Spring.
MISSED OUR ANNIVERSARY ISSUE? IT'S NEVER TOO LATE!


ras Shypka on Unsplash

HEURISTICS, WEB DESIGN

VIP BOA – A Heuristic for Testing Responsive Web Apps

EDUCATION, SPEAKING TESTER'S MIND

ISTQB, Fever Dreams and Testing

EDUCATION, WOMEN IN TESTING

How To Read A Difficult Book

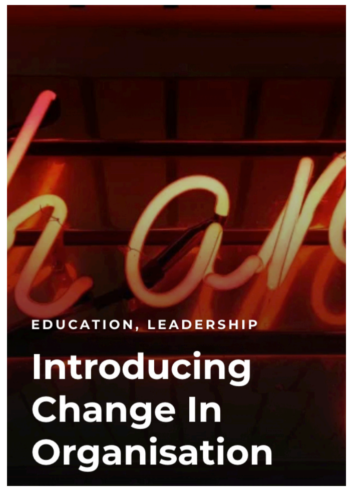
INTERVIEWS, OLD IS GOLD

Over A Cup Of Tea With Jerry Weinberg

mon Migaj on Unsplash

SPEAKING TESTER'S MIND

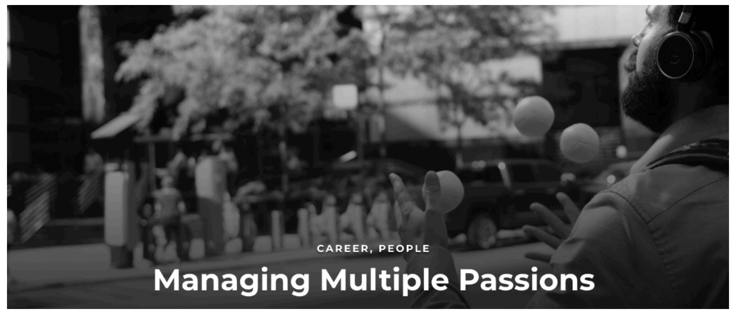
Software Testing And The Art Of Staying In Present


EDUCATION, LEADERSHIP

Introducing Change In Organisation

PEOPLE AND PROCESSES, WOMEN IN TESTING

Addressing The Risk Of Exploratory Testing Part 2


CAREER, PEOPLE

Managing Multiple Passions

LEADERSHIP, OLD IS GOLD

Leading Beyond The Scramble:

After nearly twenty years of working in software, I
many companies. One of them is what I call the sc

INFOCUS

Do you know all these amazing articles?

Great things survive the test of time.

Over the last ten years, Tea-time with Testers has published articles that did not only serve the purpose back then but are pretty much relevant even today.

With the launch of our brand new website, our team is working hard to bring all such articles back to surface and make them easily accessible for everyone.

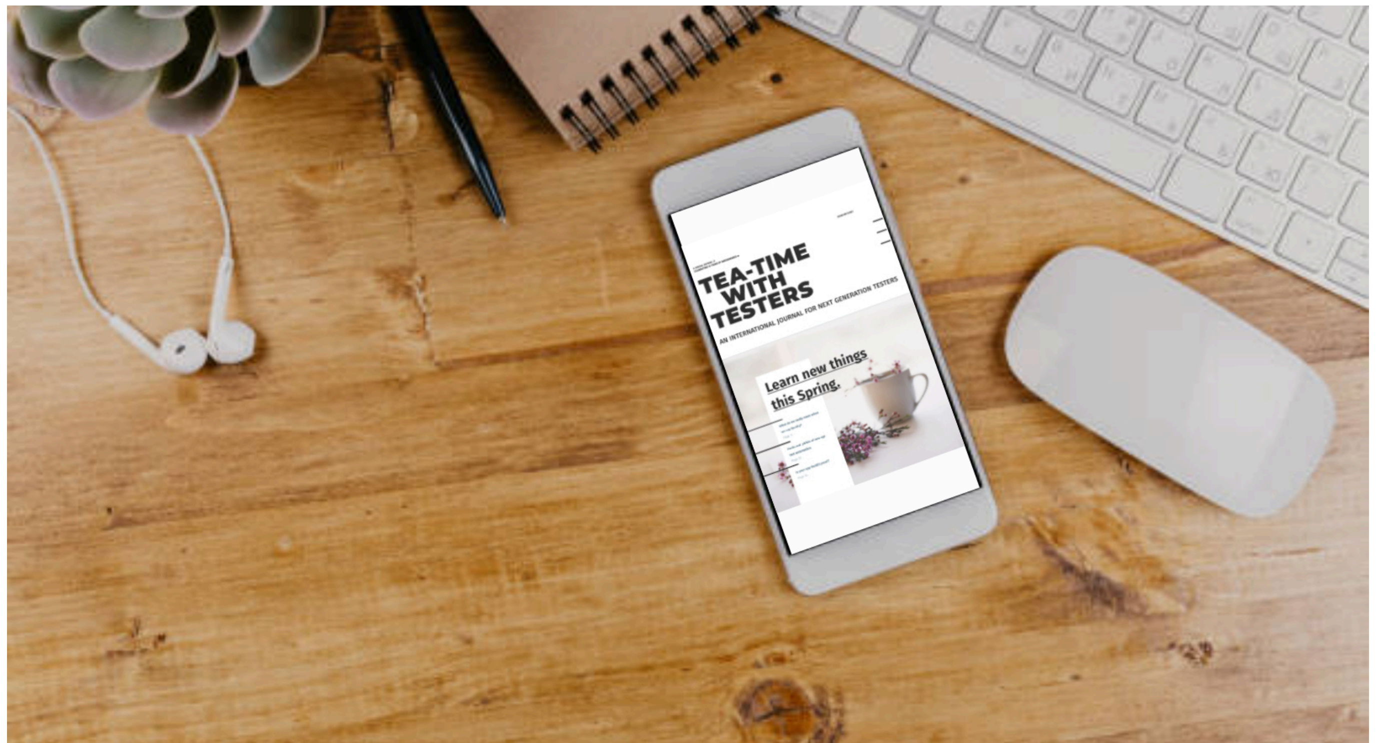
We plan to continue doing that for more articles, interviews and also for the recent issues we have published.

Visit our website www.teatimewithtesters.com and read these articles.

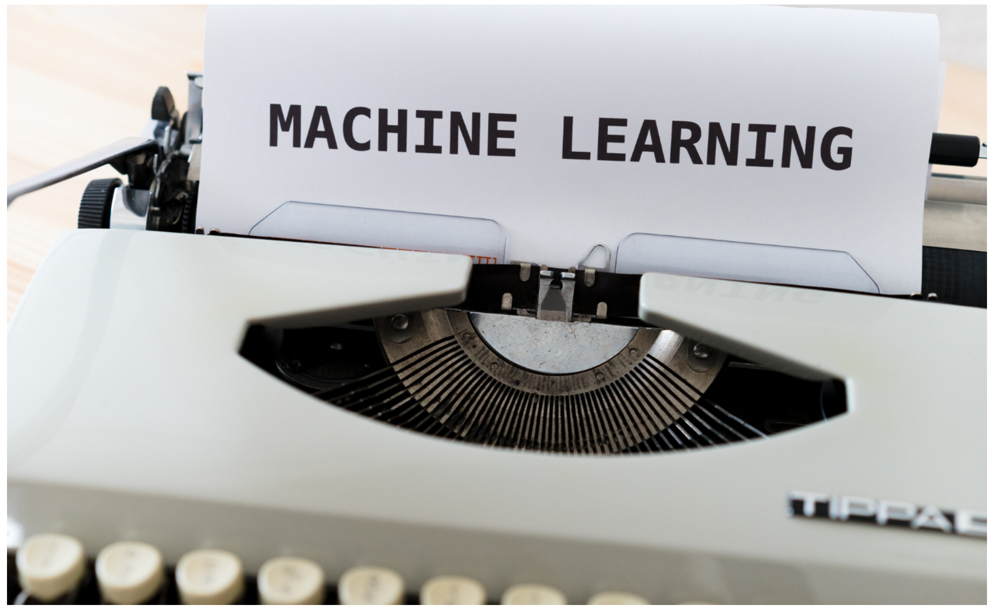
Let us know how are they helping you and even share with your friends and colleagues.

If you think we could add more articles from our previous editions, do not hesitate to let us know.

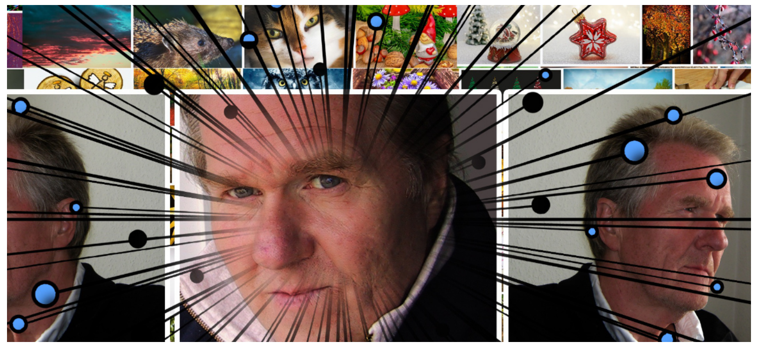
Enjoy the feast!



~Products



PART 2



MACHINE LEARNING FOR TESTERS

PAUL MAXWELL-WALTERS

–

A British software tester based in Sydney, Australia with about 10 years of experience testing in agriculture, financial services, digital media and energy consultancy. Paul is a co-chair and social media officer at the Sydney Testers Meetup Group, along with having spoken at several conferences in Australia.

Paul blogs on issues in IT and testing at http://testingrants.blogspot.com.au and tweets on testing and IT matters at @TestingRants.

Forests and Ensemble Methods

Decision trees have an intuitive, easy to follow nature, but from an accuracy point of view are more useful en masse

Lots of random decision trees can be grouped together to form "forests". These provide very high accuracy and are called "Ensemble Methods".

- **Bagging** - Each model trained with a random subset of the training set and aggregated.
- **Boosting** - Poor models summed or reweighted (a technique known as "Adaboost") to form strong models.
- **Random Forests** - uses a high number of decision trees generated from randomly selected sets of features. It is highly uninterpretable but very powerful and has generally good performance.

·

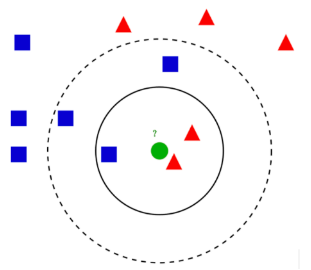
k-Nearest Neighbour

(Images from https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm)

Not to be confused with k-Means Clustering, this is a simple technique where we classify some data point or object by its (k) nearest neighbors.

- For a data point, take the nearest k items
- Whatever category has the largest % of the k items, that is deemed the category that the data belongs to.
- Do for all unclassified data

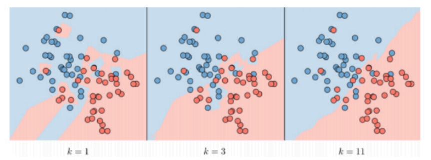
.As an example, for the blue square and red triangle data set below -

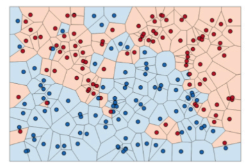


- k = 1, nearest = 1 red triangle, category = red triangle
- k = 3, nearest = 2 red triangles + 1 blue square, category = red triangle
- k = 5, nearest = 2 red triangles + 3 blue squares, category = blue square

K-Nearest Neighbours with Larger Data

An example below, shows this method applied to a larger dataset with k =1, 3, 11.





From https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/

When done for all points in training as per the previous example, we get points grouped by category.

The second image shows regions between points bounded equidistant to other points, commonly called **Voronoi Cells.** The lines that separate points of different categories are called **Decision Boundaries.**

Neural Networks

These are complex structures vaguely inspired by the biological neuron connections in our brains and are used to solve many different types of predictive and classification problems in machine learning. Each "neuron" in this case is a node that contains a series of inputs and weights along with a bias. They either output with an output to match the expected output or on output treated as an input into another layer of nodes.

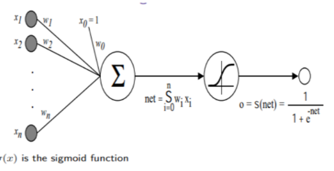
The Perceptron

In the section on Logistic Regression, we introduced the idea of binary separation a non-linear Logistic Function. We stated the logistic function as below -

$$p_i = \frac{1}{1+e^{-(\beta_0+\beta_1 x_{1,i}+\cdots+\beta_k x_{k,i})}}.$$

This is a mathematical definition of what is known as a 1 Layer Perceptron or the most basic 1 Layer Artificial Neural Network.

A perceptron is a node that takes in a set of weighted inputs, a bias, applies a logistic function and spits out an output as below


x_1
x_2
x_n
w_1
w_2
w_n
$x_0=1$
w_0
Σ
net = $\overset{n}{\underset{i=0}{S}} w_i x_i$
o = S(net) = $\frac{1}{1+e^{-net}}$

$\sigma(x)$ is the sigmoid function

$$\frac{1}{1+e^{-x}}$$

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}\cdot\mathbf{x}+b>0,\\ 0 & \text{otherwise}\end{cases}$$

where $\mathbf{w}$ is a vector of real-valued weights, $\mathbf{w}\cdot\mathbf{x}$ is the dot product $\sum_{i=1}^{m} w_i x_i$, where m is the number of inputs to the perceptron, and b is the *bias*. "

Perceptrons to Neural Networks to Deep Learning

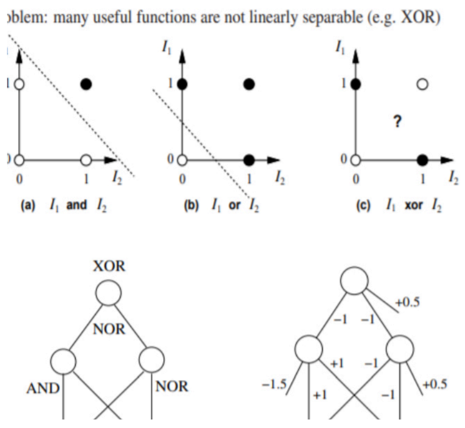
Perceptrons or single layers of perceptrons can

- classify linearly separable data sets
- do limited character recognition
- simulate some logical operators such as AND, OR, NOR etc.

but for non-linearly separable functions (i.e XOR) we need multiple "hidden" layers of perceptrons between the weighted inputs and outputs.

Networks of connected layered perceptrons are known as *"artificial neural networks".*

The use of multiple "hidden" layers in our neural networks is known as **"deep learning"**. The first multilayer neural network was designed to simulate the logical operator XOR (Exclusive OR).

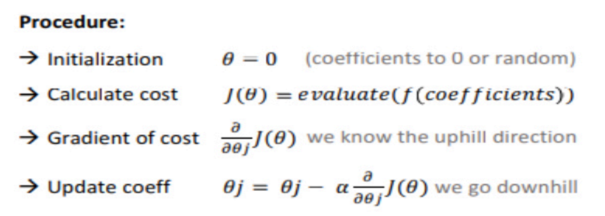


Optimising The Model Parameters - Gradient Descent

To get the best parameters for any machine learning technique you need to reduce the error (a process known as "Training"). The error is defined by a cost function, which we want to reduce. As an example, for Linear Regression this is the Mean Squared Error -

$$MSE = \frac{1}{N}\sum_{i=1}^{n}(y_i - (mx_i + b))^2$$

One way to reduce the Cost Function is to perform what is known as Gradient Descent. There are two types of Gradient Descent - **Batch** (Average the Cost across all data points) and **Stochastic** (Update the parameter for every data point).



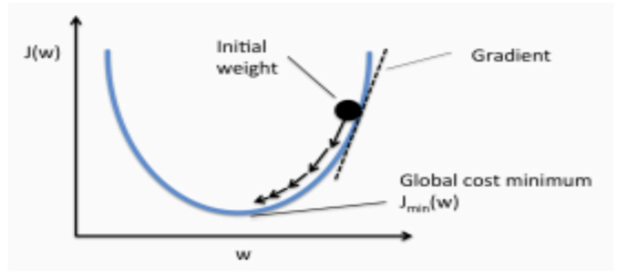
Procedure:

→ Initialization $\theta = 0$ (coefficients to 0 or random)

→ Calculate cost $J(\theta) = evaluate(f(coefficients))$

→ Gradient of cost $\frac{\partial}{\partial\theta j}J(\theta)$ we know the uphill direction

→ Update coeff $\theta j = \theta j - \alpha\frac{\partial}{\partial\theta j}J(\theta)$ we go downhill



Applying gradient descent to a perceptron linear classifier - (Equations from https://en.wikipedia.org/wiki/Perceptron)

- Set the weights to some random value or zero.
- For each example j in our training data set D do the following over the input xj and desired output dj.
- Calculate the (actual) output from your perceptron.

$$\begin{aligned} y_j(t) &= f[\mathbf{w}(t)\cdot\mathbf{x}_j] \\ &= f[w_0(t)x_{j,0} + w_1(t)x_{j,1} + w_2(t)x_{j,2} + \cdots + w_n(t)x_{j,n}] \end{aligned}$$

- Update the weights

$w_i(t+1) = w_i(t) + r\cdot(d_j - y_j(t))x_{j,i}$, for all features $0 \le i \le n$, r is the learning rate.

...until a local minimum is reached.

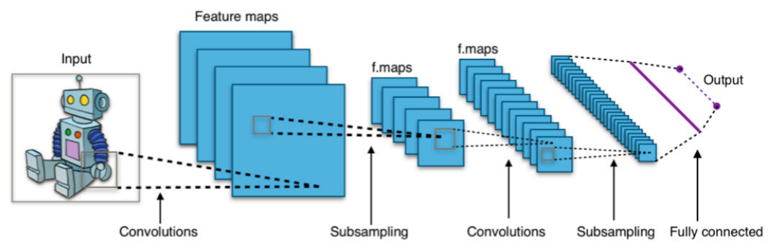
For multilayer neural networks we need to apply gradient descent to train the nodes in all the hidden layers. This is a complex process known as **Backpropagation.**

Deep Learning - Convolutional Neural Networks (CNN)

- These are complex but important multilayer neural networks with many use cases i.e.
- Image Recognition and Computer Vision
- Recommender Systems (i.e. used by Netflix etc. to recommend shows based on previous watch history)
- Natural Language Processing
- Drug Discovery (i.e. AtomNet, 2015)
- Time Series Forecasting

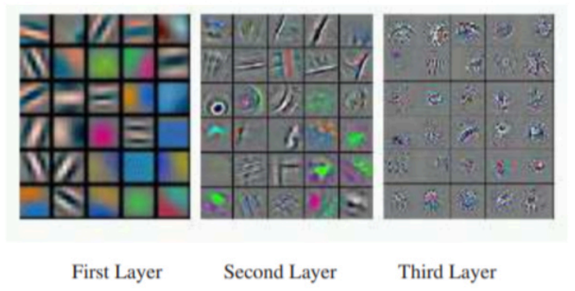
Taking the example of computer vision, to identify (say) an animal or bird from an image it is necessary to identify features (i.e. limbs, feathers etc.) that may appear more than once in the image and assign weights to them.

For this we need a neural network with hidden layers that apply some function to the inputs from the previous layer (known as a convolution) - abstracting the image in the previous layer to a map of features.



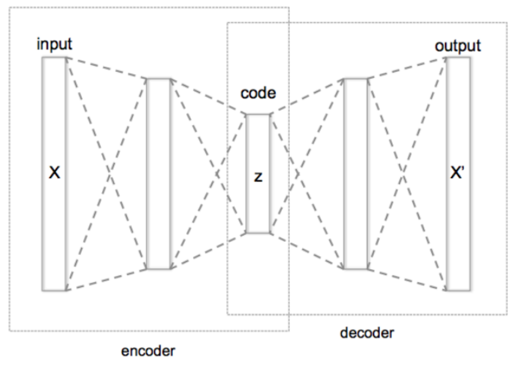
From Aphex34, https://en.wikipedia.org/wiki/Convolutional_neural_network#/media/File:Typical_cnn.png

For the example of AlexNet, a CNN created in 2012 for image recognition with 5 convolutional layers and three fully connected layers, the features of the image are abstracted to the following in each layer.



They do not seem much but from these groups of features an object in an image can be recognised.

Deep Learning - Autoencoders



From Chervinskii - https://en.wikipedia.org/wiki/Autoencoder#/media/File:Autoencoder_structure.png

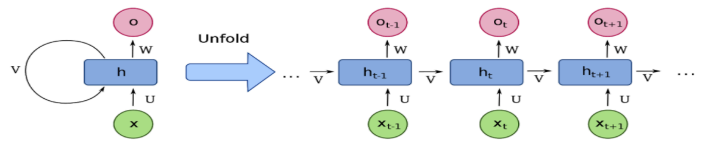
These encode an input to fit into a smaller layer, reduce noise and decode the input back to the output. This reduces the number of inputs or features ("dimensions") the model needs.

They are used in predicting social media posts, image recognition, drug discovery and (more nefariously) deep fakes.

In the deep fake above, a scene from the film Man of Steel is shown with the actor Amy Adams' face replaced by that of actor Nicholas Cage, while maintaining the same facial expressions and head movements. This can be done using autoencoders.

Deep Learning - Recurrent Neural Networks

Neural networks have been developed that have nodes form a directed graph along some temporal sequence. This makes them more dynamic (i.e. exercising an internal "memory", passing data from one layer back into the same node or earlier layers). These are called Recurrent Neural Networks.



By fdeloche, https://en.wikipedia.org/wiki/Recurrent_neural_network#/media/File:Recurrent_neural_network_unfold.svg

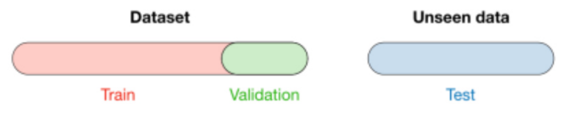
They have important applications particularly in speech recognition, natural language processing, handwriting recognition and translation.

Testing Model Selection and Performance

We have our models but how do we know if they are accurate? We have to compare their results to known output data.

Model Selection - Static Test and Validation Data

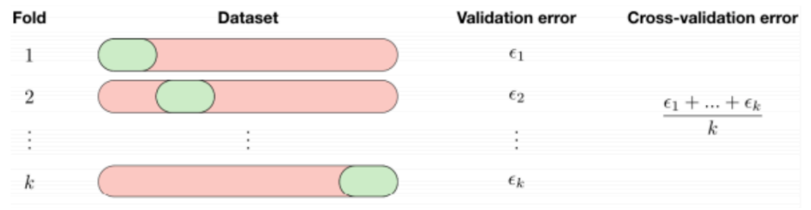
Data for training model parameters is typically split up into "training" data and "validation" data. We also have additional (separate, unseen) test data.



- **Training Data** - typically 80%
- **Validation (or "holdout") Data** - used to assess the model. Once a model passes assessment ("selection"), it is added to the training data and retrained.
- **Test Data** - only used for making predictions after the model passes validation.

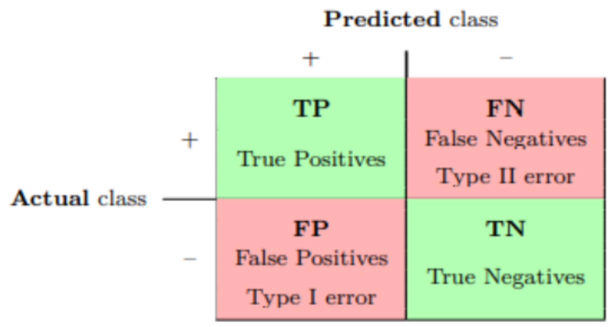
Model Selection - k-Fold Cross Validation

Another way to validate a model is to repeatedly cut all non-test data into k sections ("folds"), train on k-1 sections and validate on the final section. Then the validation errors are averaged together. This means that the validation is less dependent on a specific training data set.



Evaluating the Model - Binary Classification Performance

For binary classifications where there are only two possible categories (i.e. having an illness or not, surviving the Titanic or not), we can evaluate the model accuracy by counting all the true positives, true negatives, false positives and false negatives and creating what is known as a Confusion Matrix.



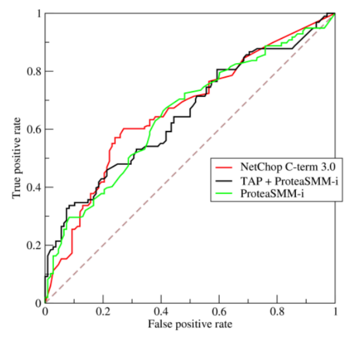
		Predicted class	
		+	–
Actual class	+	**TP** True Positives	**FN** False Negatives Type II error
	–	**FP** False Positives Type I error	**TN** True Negatives

These can be used to develop the following metrics you may come across.

- True Positive Rate (TPR) = True Positives / (True Positives + False Negatives)
- False Positive Rate (FPR) = False Negatives / (False Positives + True Negatives)

Receiver Operating Characteristic Curves

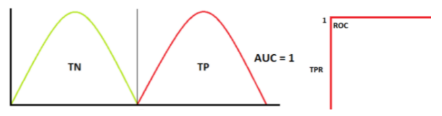
True Positive Rate (TPR) and False Positive Rate (FPR) can be plotted on a graph known as a Receiver Operating Characteristic (ROC) curve.



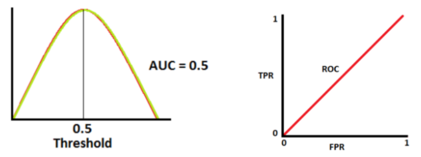
Credit BOR: https://commons.wikimedia.org/wiki/File:Roccurves.png

Interpreting ROC Curves

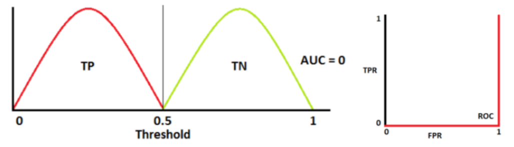
The area under the ROC curve (or just "AUC" or "AUROC") is a measure of the likelihood that the model can distinguish correctly between the two categories. (Attr - https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5)



AUC = 1, model distinguishes between each category perfectly. GOOD!!



AUC = 0.5, model has one false positive for every true positive - no better than random. BAD!!

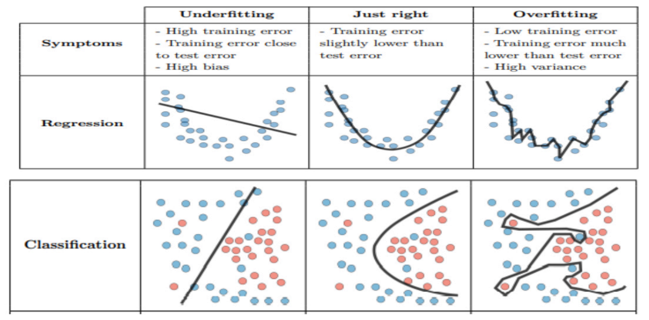


AUC = 0, model is actively mislabelling all output as the wrong category. VERY BAD!!

Model Evaluation - Bias and Variance

ML defines model predictions and outputs as having some level of bias and variance.

- **Bias** - Difference between a parameter's expected value and the actual value.
- **Variance** - Sensitivity of the expected value to small values in the training set.



	Underfitting	Just right	Overfitting
Symptoms	- High training error - Training error close to test error - High bias	- Training error slightly lower than test error	- Low training error - Training error much lower than test error - High variance
Regression			
Classification			

If the Bias is too large then the modelled relationship between input and output is poor. This is called **Underfitting**. This can be reduced by more input data, more modelling or (for k nearest neighbour) reducing k.

However if the Variance is too large then the noise in the training data is modelled too closely. This causes errors when evaluating the model against test data. This is called **Overfitting**. This can be mitigated by reducing variables ("features") in our model, (for k nearest neighbour) increasing k or specific mathematical techniques such as L2 and ridge regression.

Steps that reduce bias often increase variance and vice versa. In order to get a good model we require a balance, the need for which is known as the **Bias-Variance Tradeoff**.

Evaluating the Model - Prejudicial Bias in Data

There may be other bias from the data itself that induce discriminatory practices.

11 October, 2018 - Amazon forced to close down its AI hiring tool.

In 2014 Amazon decided to replace some of its HR functions with an AI tool which used ML techniques to screen candidate resumes and recommend candidates for interview. It allocated candidates scores of 1 to 5, however due to the historic lack of women in IT they were unfairly prejudiced in the ranking.

Areas of unfair downgrading included -

- Inclusion of the word "women" – i.e. extracurricular activities like "women's drama club leader"
- Listing attendance at one of two all-female colleges, perhaps because no past Amazon employees went to either of them
- Favouring language more commonly found on the CVs of men, such as "executed" and "captured".

In this case the bias was not in the AI tool itself but in the data. Its machine learning algorithm picked up biases from years of previous hiring practice and optimised for them.

Despite efforts to have the algorithm respond neutrally to gender biased actions in the data as listed above, Amazon executives eventually lost hope that the tool would not find other ways of discriminating against certain groups of applicants and disbanded the project. However it provides a sharp warning to future efforts to automate the hiring process.

From the Reuters article - "The company's experiment, which Reuters is first to report, offers a case study in the limitations of machine learning. It also serves as a lesson to the growing list of large companies including Hilton Worldwide Holdings Inc HLT.N and Goldman Sachs Group Inc GS.N that are looking to automate portions of the hiring process.

Some 55 percent of U.S. human resources managers said artificial intelligence, or AI, would be a regular part of their work within the next five years, according to a 2017 survey by talent software firm CareerBuilder."

(Attr: https://www.hrmonline.com.au/technology/ai-bias-can-affect-hr-just-ask-amazon/ and https://www.reuters.com/article/us-amazon-com-jobs-automation-insight/amazon-scraps-secret-ai-recruiting-tool-that-showed-bias-against-women-idUSKCN1MK08G)

Epilogue

Far from being some statistical and mathematical black box to be used by data scientists to perform predictive magic, machine learning techniques can be learnt and understood by developers and testers. QA has a role to play in the expanding area of machine learning applications and we will be working on these types of projects in the years ahead. We ignore it at our peril.

This essay, while being long and somewhat terse from a conceptual and mathematical perspective, helps to open the eyes of the testing community to the different concepts, terminologies and techniques available to execute and evaluate machine learning techniques. It points towards the types of knowledge testers will need to work in the machine learning space, and hopefully the information in this and in other machine learning resources will allow for a better dialogue between data scientists and software testers.

COMMUNITY

CAST 2021. An experience report by James Thomas.

On Monday 8th November I flew from London to Atlanta on one of the first flights permitted after the US opened up again. There were television crews crawling all over Heathrow airport, flight attendants waving the Stars and Stripes, and even someone in a Statue of Liberty costume. That was quite a thrill.

On the day I spent in downtown Atlanta I visited World of Coca Cola. While I was there I drank too much fizzy pop from around the world and told the Coca Cola Polar to grin for a selfie, which it did. That was also quite a thrill.

The conference was at Truist Park, home of the Atlanta Braves baseball team. On Monday evening we had a tour, even getting to step onto the turf and sit in the dugouts. On Tuesday we had a day of listening to knowledgeable people share their experiences of software testing, and then discussing them and comparing them to our own. That was the biggest thrill of all.

CAST 2021, my first CAST.

CAST is the Conference of the Association of Software Testing, a long-running event well-known for being a place where practitioners get together as peers. It's the conference that confers, breaking down the barrier between speaker and attendee with its Open Season after each talk. Instead of a traditional Q&A in which audience members take it in turns to ask a question or monologue about their hobby horse vaguely related to the talk, CAST has a facilitated conversation in which audience members and speakers alike are able to put their perspective and listen respectfully to those of others.

Ben Simo shared a model of testing as knowledge discovery that presents areas in which confirmatory and exploratory activities take place with a grey "horizon" separating them. Confirmatory work happens around material that is understood and exploration attacks the unknown. The model has an additional dimension stretching from stakeholder to technology which can reflect the idea that the understanding is usually not regular - often testers will be more familiar with the stack than the domain, for example. The horizon's angle and position are flexible, moving as more understanding is accumulated or as it becomes apparent that previous understanding was incorrect. He followed this up with a series of examples of ways in which automation, often confined to the confirmatory, can contribute to the exploratory testing.

Rajni Hatti spoke about how she bootstrapped security testing experience while working on a project. By using the OWASP Top Ten as an initial oracle, she was able to identify a credentials issue and perform a malicious payload injection. By the time that professional security personnel were brought in she'd learned enough to be able to interact with them as equals which enabled sharing of ideas, risk assessments, and tooling.

Laurie Sirois told us that when we laugh our colleagues laugh with us. Humour is a great way to defuse a situation, build relationships, and generally feel better about ourselves. Those of us without a funny bone shouldn't feel left out. We can just smile to improve our own and other people's moods, or share interesting observations: ha! ha! and aha! are not so different.

Greg Sypolt presented a continuous integration system for testing online exam courses. Nothing unusual there, except that it derives its test cases from models of the system. When the model changes the cases are automatically regenerated and re-run. Model-based testing is not new, but finding a non-toy system at the core of this kind of architecture is unusual. To some extent it's possible because the system under test is relatively straightforward, but a significant contribution is the commitment to making it happen and accepting pragmatic implementation choices to make it work.

Tariq King reckons that AI needs testers to prevent the world from being overrun by bad software. Sure there are challenges when starting out in AI, not least the complex statistics and the idea that oracles are likely to be fuzzy. But testers are used to working with uncertainty and learning is part-and-parcel of the craft, so we should stand up and be counted.

Angie Jones had a very flaky build. Too many tests were failing too much of the time, the deployment pipeline was backed up, and trust had been lost. Her team worked themselves out of the deep hole through a number of initiatives, including separating out the tests that failed often into a non-blocking suite; assigning a test monitor; making tickets for fixing each flaky test so that they could be prioritised alongside other work; and imposing a limit on the proportion of tests that were permitted to be failing before all other work stops.

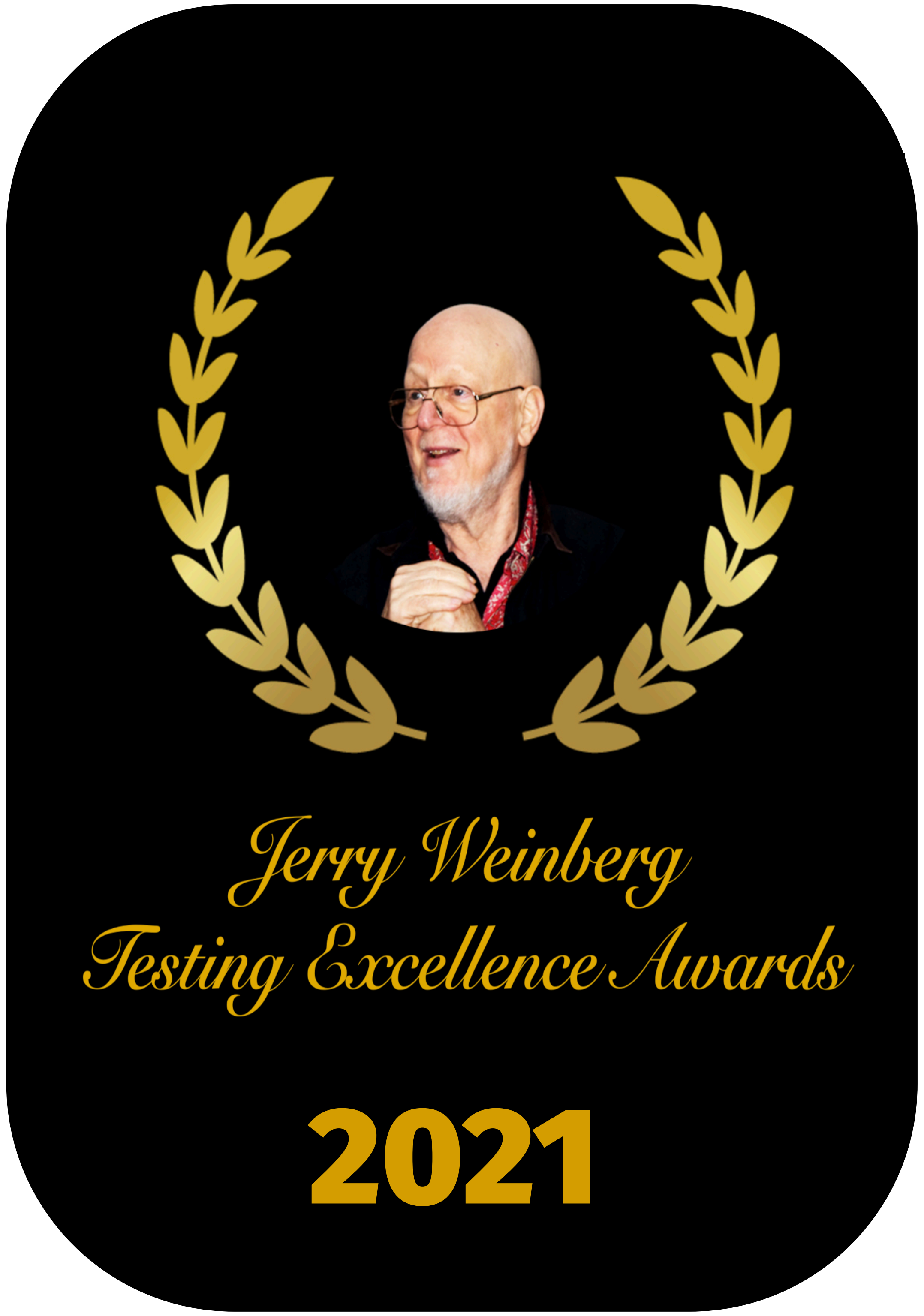
I had a blast at CAST. It had the vibe I'd been wanting and expecting despite being a Covid-safe event and sized for just 50 people. Planning for next year is already underway. I expect to be there and I hope you can make it too.

See also:

My sketchnotes from the conference

Chris Kenst's video diary

~Awards



Tea-time with Testers is immensely pleased to make this announcement. This is something we have been wanting to do for quite some years and we are glad that we are finally doing it.

We are delighted to make yet another contribution to the testing community. And this time it is software testing awards.

Why new awards while there are already quite some?

The better question is, why not?

Actually, that's the answer we got for ourselves when we decided to start Tea-time with Testers while there were already quite some magazines on market. We listened to our inner calling and did what we felt was the right thing to do. And so here we are, doing it once again with the software testing awards.

Presenting "Jerry Weinberg Testing Excellence" awards.

If you are our regular reader then you might be aware of the long association we had with Jerry and how much influence he had on what we delivered over the last decade.

What could be the better way to honour Jerry and celebrate good testing together?

Through his feedback and regular reviews of the issues we published, Jerry taught us the importance of recognising good work and creating a platform to promote good testing.

Through these awards, we want to recognise and reward worthy people who we think deserve to be known by the global testing community. So that it inspires actions and encourages others in taking the craft of testing ahead.

What is the nature of these awards?

We have made some categories for this year and plan on to add few more going forward. A person's work and contribution to the testing community would be our key criteria. Our team has worked hard to make these awards as special and worthy as we could.

We will announce these awards every year in our December issue. The awardees will get a memento that we will send them via post. The most important part is to recognise these people and help their ideas reach more people. And we are committed to delivering on that.

Please join is in congratulating all the awardees for year 2021. You shall find their names and accomplishments on following pages.

Once again, a very special thanks to Jerry's wife and a great companion Dani Weinberg who saw value in this initiative and has been a great supporter of this idea.

Jerry was very generous with his time to help smart people go ahead in life. We were privileged to be gifted with his association with us. And these awards in his name are our humble efforts to pay it forward.

We are thrilled to end an year on this exciting note and super excited to see you all joining this celebration!

Sincerely yours,

Lalit

Jerry Weinberg Testing Excellence Awards 2021

– RISING STAR OF THE YEAR –

RAHUL PARWAL

When you are finished learning, you are finished as a tester. - that's the message Jerry had for the testing community when we interviewed him.

Rahul is a passionate learner. In fact, his passion for learning is highly infectious. Name the testing event worth noting and you shall find Rahul either attending it, participating in contests or being part of the event/activity itself. Rahul is one of its kind tester we came across this year who is so eager to read, learn new things about testing and who does not stop right there. He also makes sure to share his learning with community at the large.

In year 2021 alone, Rahul has published **8 e-books** through which he further shares his own learning from various sources.

In year 2021 alone, Rahul has **won 12 testing competitions and hackathons.**

In year 2021 alone, Rahul has actively contributed in organizing various testing events.

It requires uncommon amount of passion for the craft of testing, a will to excel and inspire others to do the same by leading through an example. Rahul is a rising star and we would like to see more testers like him taking the craft of testing ahead.

Congratulations Rahul. Thank you for your work and we wish to see you rising even higher.

– TESTING LEADER OF THE YEAR –

DAN ASHBY

Testing community and industry as a whole has lots of great testers, great thought leaders and philosophers. But the business leaders who understand testing, study the craft of testing, contribute to the advancements of the field and represent meaningful testing in organizations are just handful.

We firmly believe that, driving a great culture of quality and testing within organizations requires a strong and effective business leader. A leader who knows what meaningful testing looks like, a leader who knows how much to push the test engineering pedal and where to encourage creativity and craftsmanship in testing. A hands-on leader who immerses themselves in the changing contexts and develop solutions to create value through testing, instead of blindly buying industry best practices(?) that add little or no value.

Dan Ashby is one such leader that testing industry has always needed. His passion for taking craft of testing ahead is inspiring and so is his dedication to building new models for testing that are time relevant and context appropriate. His popular model of "**Continuous Testing in DevOps**" quite effectively saved "testing" from getting lost in the process or getting too thinly spread to be even noticed.

And that's not it, Dan has been inspiring actions and making significant contribution to the testing field by **doing everything** he does.

This award for **Testing Leader of The Year** had to go to Dan, hands down! And we need more leaders like him.

Congratulations Dan. Thank you for leading the way.

Jerry Weinberg Testing Excellence Awards 2021

– (THINKING) TESTER OF THE YEAR –

FILIPE FREIRE

There are two kinds of excellent testers in this world. Those who think critically, and

Having testers like Filipe in the community is more rewarding for community itself, we must say. Filipe caught our attention with his earnest and sincere piece of writing i.e. **Self proclaimed UN of Testers latest discovery**. In fact, everything that he writes has deep substance to it.

The face and fate of professional testing could be much different today should we have had a lot many testers like Filipe who think critically, who study the craft and most importantly who take the stand for meaningful testing. We believe it is more important to think critically and say no to bad testing than embracing best practices for good testing without questioning them at all.

By asking questions that matter, by challenging the so-called best practices, by finding alternatives that are effective, by being technically sound yet highlighting the importance of critical thinking, Filipe has become a true role model for testers of present times. And we believe the “Model Filipe” is built to excel in all changing contexts.

If you have not yet come across **Filipe and his work** we highly recommend you to do so.

Congratulations Filipe. Please continue to help us see the hidden ~~costs~~ side of things.

– TESTING INSPIRATION OF THE YEAR –

ANNA ROYZMAN

There is an interesting thing about stars in the sky. Have you noticed how some of them rise and shine for a while then disappear from the horizon and how some of them consistently appear for all year round? Which stars would you rely on if you are to seek guidance and direction when lost?

Since our own inception as a publication, we have closely observed how testing community and the craft has evolved throughout the decade. Anna Royzman is one such star who has been consistently shining, is firmly positioned, always shows up and has been an inspiration for many.

From being a hands-on test practitioner to inspiring testing leader, from being a conference speaker to creating conferences that make a difference, from being part of the community to being a community builder, from taking stand for the craft to contributing to the advancement of the craft, Anna has done it all her entire career and she continues to do so.

Anna’s **entire career journey** has been filled with stories of determination, passion, conviction, craftsmanship and courage. She is truly an inspiration and we thank her for being who she is.

Congratulations Anna. Thank you for your contribution the field and the difference you have made.

TEA-TIME WITH TESTERS

JOURNAL FOR NEXT GENERATION TESTERS

CONTENT PREVIEW : ISSUE 01/2022

READY TO ROCK THE NEW YEAR 2022? WE CERTAINLY ARE!

INTERNATIONAL JOURNAL FOR NEXT GENERATION TESTERS

TEA-TIME WITH TESTERS

THE SOFTWARE TESTING AND QUALITY MAGAZINE

TEA-TIME WITH TESTERS
AN INTERNATIONAL JOURNAL FOR NEXT GENERATION TESTERS
Learn new things this Spring.

Created and Published by:

Tea-time with Testers.

Schloßstraße 41, Tremsbüttel

22967, Germany

Editorial and Advertising Enquiries:

- editor@teatimewithtesters.com
- sales@teatimewithtesters.com

Be with us and visit our website:

www.teatimewithtesters.com