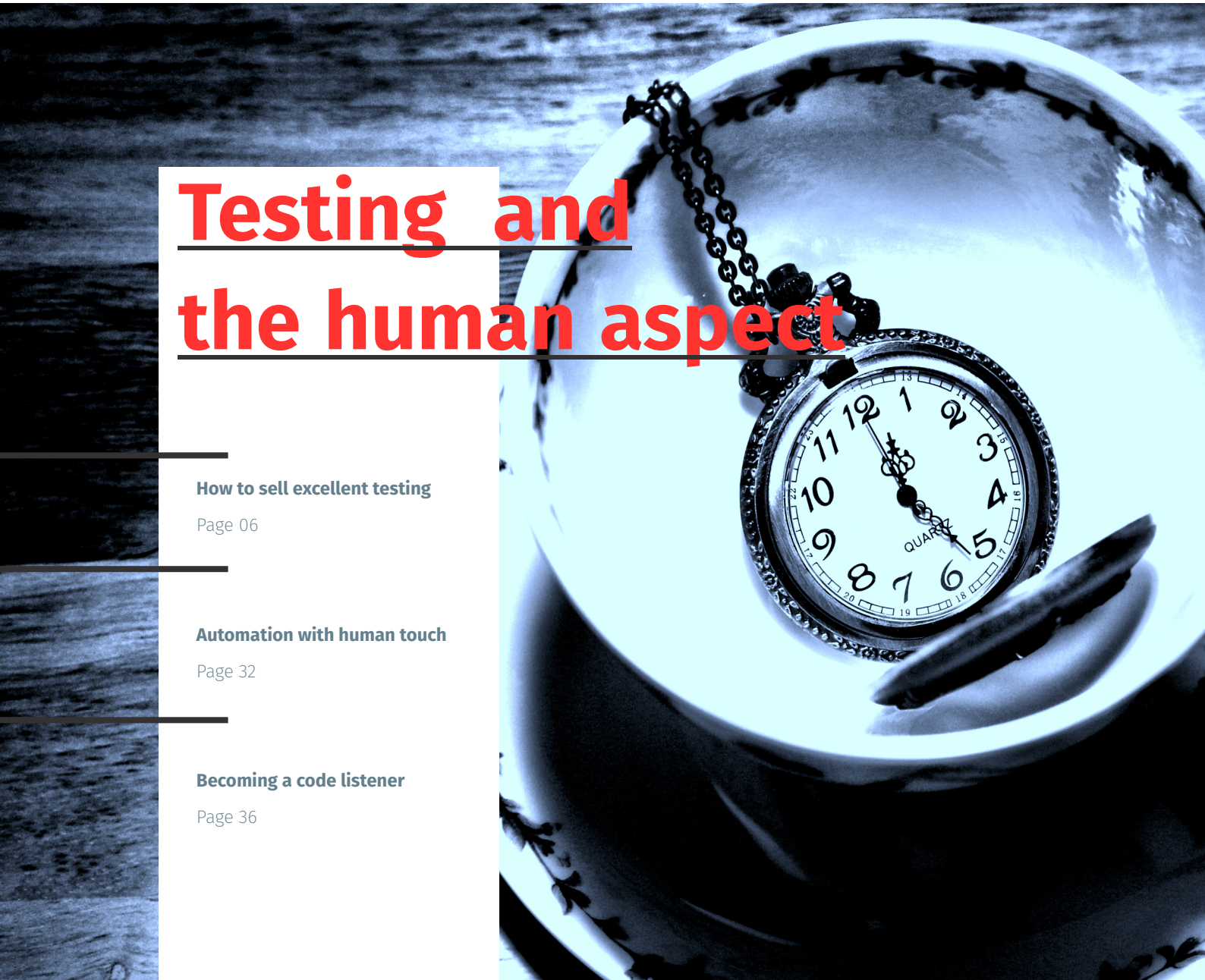
WAKING TESTERS UP SINCE 2011

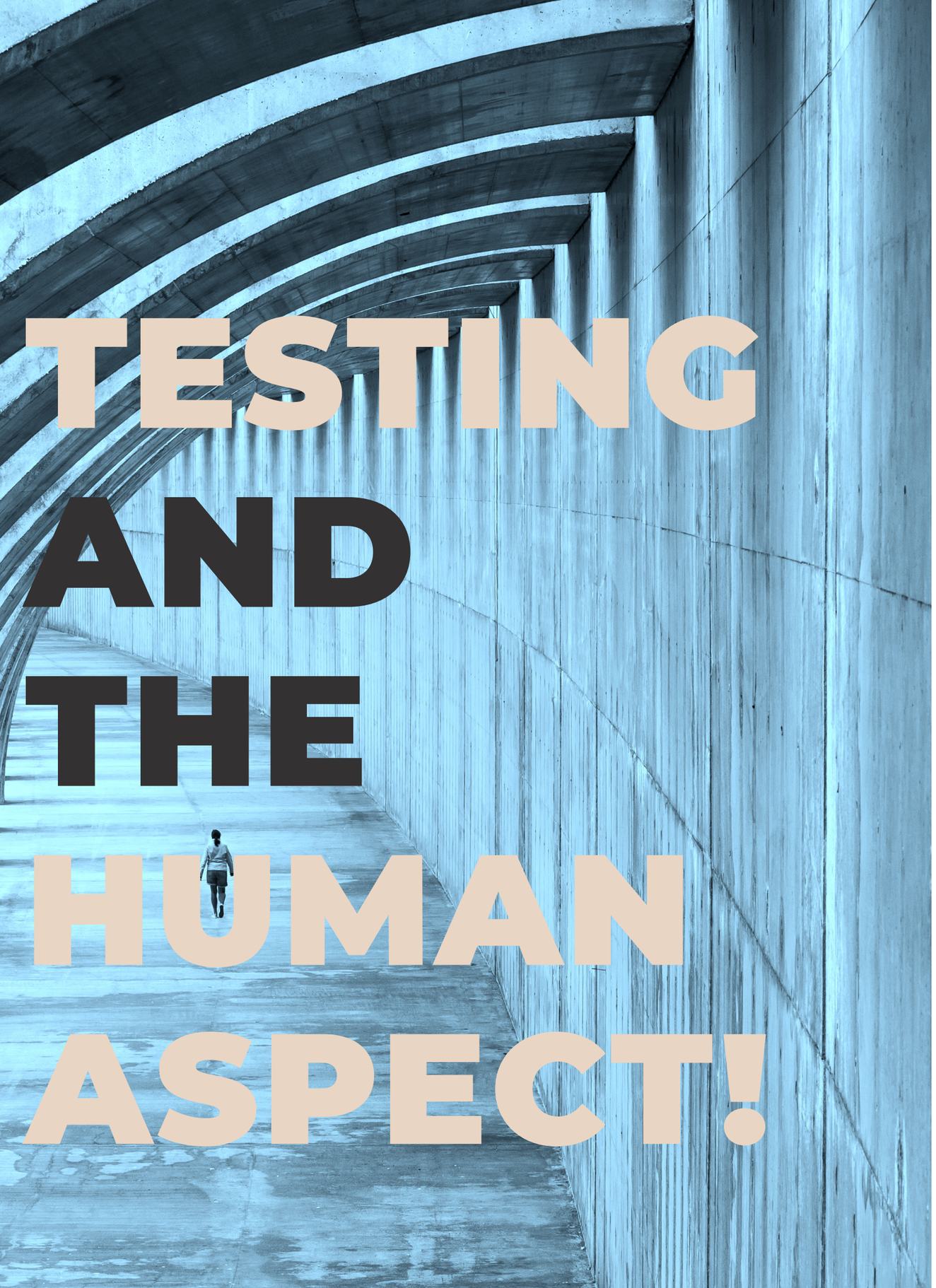
ISSUE #01/2022

TEA-TIME WITH TESTERS

AN INTERNATIONAL JOURNAL FOR NEXT GENERATION TESTERS



Testing and the human aspect

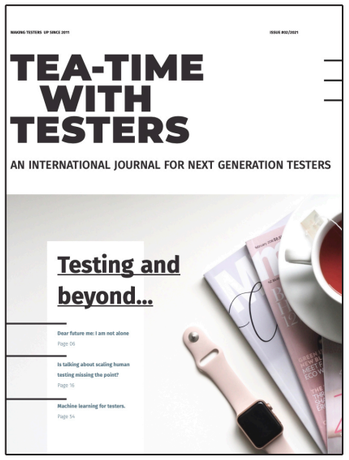


TEA-TIME WITH TESTERS

EDITORIAL BY LALIT

INTERVIEW: 26-30
A CUP OF TEA WITH JAMES THOMAS

TEA-TIME WITH TESTERS



A NEXT GENERATION MAGAZINE

FULL OF CONTENT AND TIPS FOR TESTERS

It's the little things...

If you ask me what has been my personal highlight of the year so far, I would say it's EuroSTAR conference that happened in Copenhagen this year.

Was it because my talk was selected for this conference? No. Was it because my submission received the coveted EuroSTAR Best Paper 2022 award? Not really! I mean, it is indeed an honor and I consider it one of the biggest accomplishments in my testing career so far.

But there was one little thing that happened while I was at EuroSTAR. And that thing has made me feel beyond humbled. It's something I would keep myself reminding of whenever I question myself if Tea-time with Testers should still be up and running.

It would be a lie if I tell you that I never thought of discontinuing the magazine. My reasons and rationale behind thinking so are secondary. But one little conversation I had with a fellow speaker at EuroSTAR conference, changed my whole perspective about Tea-time with Testers, and the purpose a magazine can serve.

I happened to have a conversation with a passionate tester, conference speaker, a great mind who is also a Head of QA at a technology firm. She told me some interesting stories about how much she loved reading Tea-time with Testers and how things she learned from the magazine have helped in her professional career. There was a time when she was allowed to print only two pages per day at her work place. So she would ask some of her colleagues to print two pages each for her. That way, she she got the whole magazine printed so she could continue reading at home and learn from different articles published. She acknowledged the role this magazine has played in her growth as a tester and where she is in her career today.

Such stories and acknowledgments coming from people I have great respect for makes me feel immensely proud and honored at the same time. For me, this has been the true reward for all the work entire team at Tea-time with Testers has done over the years.

This little conversation has given me another reason to keep the magazine up and alive, for you never know how it will help someone else's career as a tester. And as luck may have it, we are getting new passionate members reaching out and joining our team.

I am pleased to announce Dave Levitt as new member in our editorial team and little by little we plan on to add more members in Tea-time with Testers family. More awesomeness is on your way.

I thank you for your patience with the little pause we took this year.

After all it's little the things that make big difference. Don't they?

LALITKUMAR BHAMARE
Chief Editor "Tea-time with Testers"
–
Manager - Accenture Song, Germany
Director - Association for Software Testing
International Keynote speaker.
Award winning testing thought leader.
Software Testing/Quality Coach.

Connect on Twitter @Lalitbhamare or on LinkedIn

“HOW TO SELL EXCELLENT TESTING?”

INGO PHILIPP

- Ingo is a seasoned product management professional who helps companies to develop, marketize, and sell software products & services that people need, want, and will pay for.

He holds a master's degree (MSc) in astrophysics, an MBA from the Vienna University of Economics & Business, and has deep technical skills from 10+ years of experience in software engineering.

In his last positions, his responsibilities ranged from product engineering, product management, marketing (e.g., product, event, content) to sales, pre-sales, and evangelism.

This article shares 8 principles for selling the value of testing strategically to make people in your organization perceive excellent testing as what it is: a value center, not a cost center.

Introduction

Even though software development is so dependent on software testing, hardly anyone in management understands anything about it. Testing is often either willfully ignored, treated as something that could be replaced by machines, or seen as the number one bottleneck.

On top of that, excellence in testing is often confused with excellence in automation. What too many managers view as testing is not what it actually is. This lack of understanding and the resulting misconceptions make it hard to sell the value of excellent testing.

“Doing excellent testing is one thing, selling it to management is another.”

Before you start selling your testing excellence, you should know what it is. Excellent testing is hard to describe. You only find out what excellent testing is when you see it.

“If you cannot tell your value, you cannot sell your value.”

My article on excellent software testing outlines what I have seen. It boils the software testing excellence of a tester called Alice down to 18 characteristics. Feel free to use this article as an inspiration to work out your notion of testing excellence.

Here's a teeny-weeny summary of her testing excellence. Alice was our yardstick to assess software quality. She was our microscope for the actual problems in our software. She was our telescope for the potential problems in our software. She was our wake-up call for our unawareness about numerous things. She was our stop sign for our over-focus on even more things. And she was our alarm device for all the problems in our thinking.

Alice was, and most likely still is, a cocktail of testing excellence. Her testing was fast, inexpensive, credible, and accountable. It was excellent.

This article continues the story of Alice. It outlines 8 principles that aim to help you to sell testing in a strategic way. These principles will help you to change people's perception of testing in such a way that testing is no longer seen as a cost center but as a value center.

Motivation

Working with Alice has been a double-edged experience.

One, it has been a humbling experience. I have learned that I am not even close but rather a world away from being an excellent tester. Two, it has been a sobering experience. I have learned that testers doing excellent testing aren't necessarily excellent in selling testing.

“Being excellent in ding testing does not imply being excellent in selling testing.”

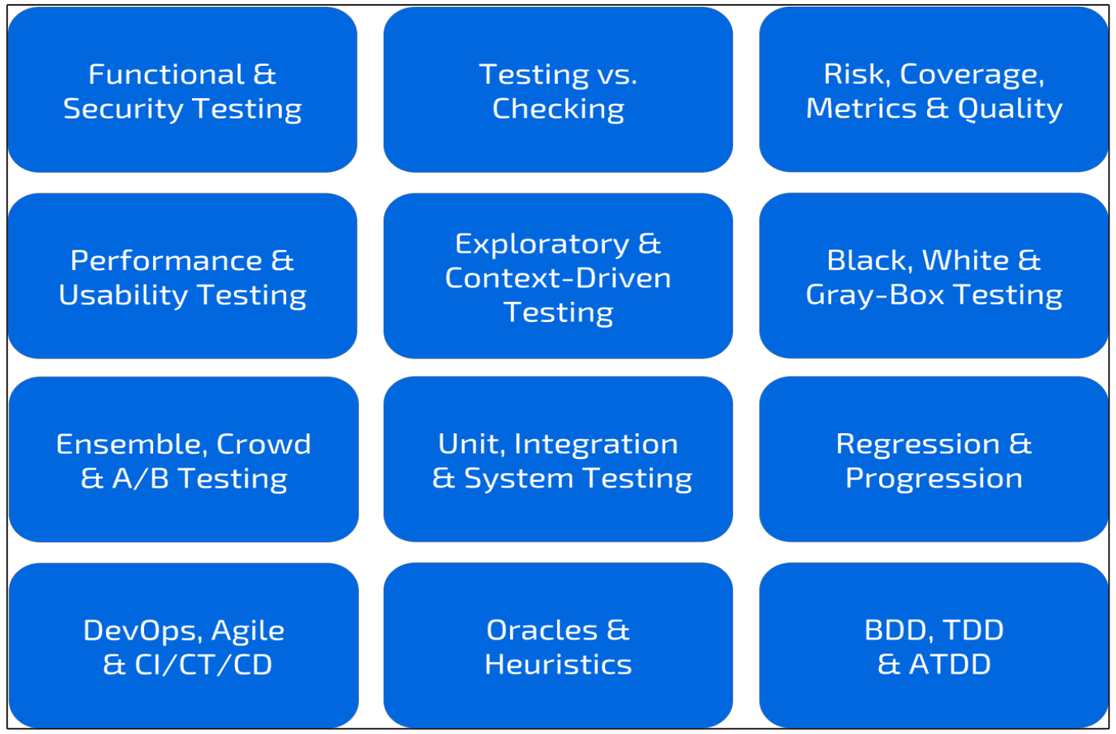
Selling testing means communicating the value of testing in a strategic way so that testing is no longer perceived as a cost center but as a value center by the people who matter.

lice was brilliant in what she did. She quickly got promoted. She became the first test lead in her company. In this role, she was responsible for mentoring and coaching more than 20 testers worldwide. Alice was used to doing testing, not talking about testing.

She was used to showing the value of her work to people inside her home territory (e.g., UI/UX experts, software developers, product owners) by doing it.

A week after her promotion, she was asked to communicate the value of testing to people outside her home territory. In this session, people from an operational level and strategic level in Alice's organization were invited. This included people from c-level management (e.g., CPO, CTO, CXO) and various directors and (senior) vice presidents.

Alice blew on the fire.



She talked about functional testing, security testing, usability testing, load testing, stress testing, and performance testing. She philosophized about the relation between testing and checking. She deep-dived into exploratory testing and session-based test management.

She spoke about unit testing, integration testing, system testing, and user acceptance testing. She gave a lecture on oracles, heuristics, risk, coverage, metrics, and software quality.

She highlighted the differences between black-box testing, white-box testing, and gray-box testing. She outlined regression testing and progression testing. She touched upon test-driven and behavior-driven development and their relation to software testing.

She discussed the potential negative consequences of false positives and false negatives. She pondered on ensemble testing, tour-based testing, crowd-testing, A/B testing, context-driven testing, and testing in production in the context of DevOps, Agile, and CI/CD.

It was a firework of testing that lasted for about 60 minutes.

The good news is that Alice brilliantly demonstrated the complex nature of software testing. She showed that software testing is something that cannot be done by anyone just like that. Her performance showed that professional testing requires special skills.

"Focus on the value of testing and not just the practice of testing. "

The bad news is that this wasn't what the audience was looking for. She sleepwalked into leaving the audience in the dark of understanding the value of testing. She missed reading the room. She missed tailoring her message to the audience.

She talked about how testing is done, not why testing needs to be done. She over-focused on the practice of testing, not on the value of testing. In her own words: "The pitch failed."

To put it mildly, Alice felt miserable after this session. However, she quickly started to take action. She drummed up her fellow testers and organized several brainstorming sessions to work out a set of basic principles for selling software testing.

These principles can be understood as our general beliefs that guided our selling behavior. They guided the way we communicated the value of testing in Alice's organization.

Principle of Selling

The common perception of selling and salespeople is all too often linked with used cars and timeshare apartments: Hit them hard, fast, and get out of town with your commission w/o a long-term thought for the customer.

Therefore, the buying process is often one of mistrust and skepticism. In our brainstorming sessions, we have learned that this distorted perception about selling made some testers feel icky, cheesy, or sometimes even sleazy when selling appeared on the agenda.

Alice's fellow testers often started painting the stereotypical picture of an egocentric, dodgy, dishonest, and money-grubbing person when the term selling was vocalized in our sessions. Their impression was that selling means convincing people in a superficial and pushy way to do something that isn't worth doing. That's not sales, that's bad sales.

Selling isn't telling. It means exchanging values. A sales transaction is an exchange of values between a buyer and a seller. The seller gives something of value (e.g., product, service) to the buyer, and the buyer, in return, gives something of value (e.g., money, data) to the seller.

The way the seller can create value for the buyer is by providing a solution to the buyer's problems. We consider a problem as a gap between what is perceived and what is desired.

Thus, there's a problem when there's something blocking the buyer from getting closer to their desired states (e.g., needs, goals, objectives, wants, desires).

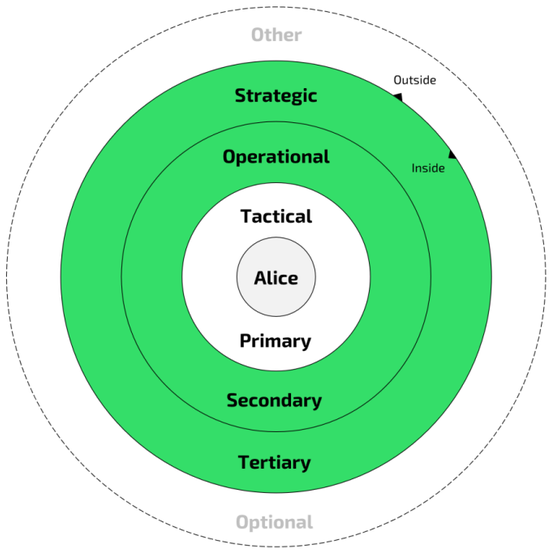
This principle reminded us of the following:

- We need to know our buyers. We need to know to whom we are selling.
- We need to understand the desired states of our buyers.
- We need to know the value we create through testing.
- We need to know how to tie this value to the buyer's desired states.
- We need to know how and when to articulate this value exchange.

This principle is a foundational one. It guided our discussions and helped us to understand that selling means demonstrating the way we create value for others through testing.

Principle of Targeting

We decided to put our buyers, our targets, into two categories: People inside and people outside software development. Inside software development, we had three groups. These groups reflected different levels of decision-making power in Alice's organization.



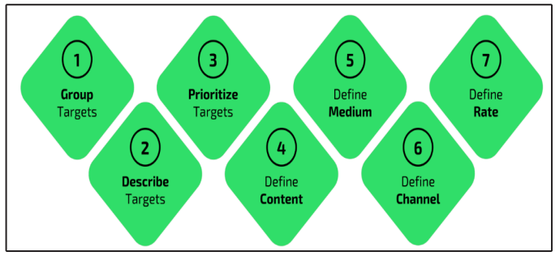
“When you speak to everyone, you end up speaking to no one.”

The term management was an umbrella term that referred to our secondary and tertiary targets. These people were the "money people" in Alice's organization. They controlled the budget of the organization, including the budget spend on software testing.

You might now wonder why the managers, with their high decision-making power, weren't our primary targets? The reason is twofold. One, these people were hard to reach. Two, their decisions about software testing were highly influenced by the people on the tactical level.

Our approach was to primarily, not exclusively, sell the value of testing through the people in our natural environment. In sales jargon, our goal was to grow the people on the tactical level to our sales champions who sell the value of testing on our behalf to the operational and strategic levels. This was important, especially when we weren't there.

We first grouped our targets according to their decision-making power. Then we developed user personas for our targets to better understand their habits, objectives, needs, and goals. Then we prioritized our targets since we cannot please all our targets all the time.



We then decided what type of content (e.g., factsheets, white-papers, infographics, success stories) we need to create to attract each target persona. Then we defined which mediums (e.g., webinars, blogs, podcasts, workshops, meetups) to use to communicate the content to the targets. Then we defined which channels to utilize (e.g., Slack, Confluence, MS Teams, email) to distribute the content. Finally, we decided how often to reach out to our targets.

In a sense, we conducted internal marketing campaigns to communicate the value of testing. In doing so, we defined metrics such as content engagements (e.g., likes, comments, shares), impressions, and audience growth to later be able to assess the success of our campaigns.

This was our basic scheme for selling testing. The challenge was not so much to create this content but to ensure that this content is used and consumed in a consistent way throughout the entire organization. This leads us to our principle of consistency.

Principle of Consistency

Your superpower is consistency.

We are stronger together than we are alone. Alice understood this very well. She started to unite her power with the power of her fellow testers. She understood that the best (testing) teams not only have chemistry but also consistency.

Alice wasn't alone. Many of her fellow testers faced the problem of selling the value of testing well. Some testers were simply better at hiding this problem than others. So, Alice quickly turned the brainstorming sessions into a regular series of meetups. Our so-called Action Lab was born. You can think of it as a community of practice.

Our Action Lab was a group of people (e.g., testers, developers) who came together on a regular basis (e.g., bi-weekly) to share their passion for testing by exchanging ideas, sharing experiences, and spreading knowledge. Everyone was welcome but no one was safe to just lean back and relax. It was a working group. The focus was on doing, not on talking.

“An ounce of practice is worth more than tons of theory”

The hands-on character of these sessions was key to keeping the participants engaged. The individual sessions had no fixed duration but usually lasted no more than two hours. Each session always focused on one specific topic that was neither too big nor too small.

The topics were phrased as questions, not as statements because questions require answers, statements do not. In our first session, we tackled the question "What characterizes testing excellence?". From this, we derived the 18 characteristics of excellent testing. In another session, we discussed "How to structure note-taking during exploratory testing?".

Ergo, the topics were not only focused on the practice of testing but also on the value of testing. The overall goal was twofold. One, to improve our daily testing practice. Two, to improve the way we communicate the value of testing. The content resulting from these sessions was then used for our internal marketing campaigns.

Each session was moderated. The moderator introduced the topic, prepared the agenda, moderated the discussion, collected action items, and summarized the lessons we have learned after each session. That's, in a nutshell, how we rolled our Action Lab.

“Consistent action is what creates consistent results.”

Our Action Lab not only helped us to close skill gaps and knowledge gaps among testers in different teams and different departments but also enabled us to realize and understand that unity is a strength and division is a weakness.

In the early days, our main learning was that even in one single organization many testers often have different, and sometimes even contrary, views on testing. Therefore, by getting together we were able to develop a consistent view on testing. This, in turn, enabled us to develop a consistent way of positioning and messaging the value of testing.

This was crucial. Otherwise, we would have signified nothing right from the beginning. To conclude, your superpower is your team's consistency, not your individual brilliance.

Principle of Practicing

Your pitch is only as good as your practice.

In addition to our Action Lab, we also organized practice sessions to train the way we pitch the value of testing. We decided to train because we have learned that the delivery of our message is every bit as important as its content.

This gave rise to our Sales Lab. We did that for three reasons. First, we wanted to perfect the way we pitch the value of testing in less than 2, 5, 15, and 30 minutes. We mainly focused on making our pitch succinct, since, in most cases, we only had a few minutes to get our point across. Therefore, focus and momentum were our bosom buddies in pitching testing.

Secondly, we have seen that some testers were deeply convinced that one must be born with the magic ability to sell well. Well, that's simply not true. Anyone can learn how to sell. You just need to be willing to invest effort. Unsurprisingly, practicing the act of selling is a great way to get released from this popular misconception. The more often you do it, the easier it will get. Selling isn't talent. Just as testing, selling is a set of skills that can be learned.

You can’t hire someone to practice for you.

Thirdly, we have learned that some testers were frightened of failing at selling. They were scared of giving the rest of the organization the final proof that they don't understand their profession or that they aren't made for success. This fear often created a mental block.

Ergo, we practiced. We practiced a lot. We practiced together to release ourselves and our fellow testers from this anxiety. As a result, these sessions boosted our confidence.

We didn't just share our success stories but also our failure stories to help our fellow testers realize that failing is not bad, not learning from failure is. This then stimulated them to speak about their failures more openly. Ergo, we learned from each other and realized that the most frightening monster isn't the one you are selling to, it's the one that exists in your mind.

Without being committed to sell well, you will never start selling well.

All this takes effort. But remember, without your individual commitment you'll never start selling well. And without your team's consistency, you'll never finish selling well.

We have learned that consistency comes from collaboration. Hence, we decided to foster collaboration through sharing knowledge. This leads us to our principle of collaboration.

Principle of Collaboration

Great selling comes from great collaboration

The true power of becoming and remaining excellent in doing and selling testing comes from sharing knowledge, not withholding it. So, we created a knowledge base for testing.

In this knowledge base, we shared the lessons we have learned from our internal seminars, workshops, and meetups (e.g., action lab, sales lab). We shared testing practices that turned out to be valuable in our projects (e.g., heuristics, charters, mind-maps, models, tools).

We created a glossary to make testing-specific terminology (e.g., context-driven testing, exploratory testing) and testing-related terminology (e.g., quality, coverage, risk) more consistent within the organization. We shared what we have learned from reading books, articles, papers, and blog posts. We shared answers to questions we frequently discussed (e.g., "How to decide what to automate?"). We shared educational material on testing such as blogs, books, magazines, courses, podcasts, reports, and conference talks.

This list could go on and on. You get the point. This knowledge base was our one-stop-shop for all content related to software testing. Therefore, we called it the Testing Shop..

“Just as great testing,great selling comes from great collaboration”

This knowledge base wasn't just a storing platform but rather a collaboration platform that enabled us to gather, incorporate, and share feedback in the wink of an eye.

This platform was based on SharePoint and Confluence to make the information accessible for everyone in the organization. This allowed us to easily involve other people too. For example, professionals in branding, UI/UX design, marketing, and sales. These people initially supported us in structuring our positioning material in a more professional form.

We created a variety of collaterals such as factsheets, one-pagers, value cards, battle cards, and storybooks. This enabled us to share our message in a more concise, to-the-point, and engaging way. So, just as great testing, great selling comes from great collaboration.

In the course of creating and sharing this content, we not only sharpened our understanding of the practical dimension of software testing, but we also sharpened our understanding of the value we create for other people through software testing.

Principle of Connectivity

Make your story their story

Having a crystal-clear understanding of the value of testing was vitally important since we usually received two types of questions from management. The first one was: "Why do we need testing at all?". We addressed this question in the following way.

First, we made it clear to management that the value we provide through testing is largely intangible. You cannot touch it. Through testing, we collect quality-related information (e.g., risks) about the software to enable other people (e.g., developers, product owners) to make better-informed decisions (e.g., shipping decisions, fixing decisions). We inform people.

Next, we let our management know that problems are the quality-related information we primarily are looking for in the software. Because making these people aware of problems means enabling them to address these problems. And being able to address these problems means being able to avoid that these problems turn into bigger problems.

So, giving people the ability to avoid problems means giving them the ability to mitigate potential damage, e.g., in terms of financial loss, loss of reputation, or the loss of faith of clients due to poor user experience. Simply put, through testing, we help other people to mitigate risk by making them aware of the risks (i.e., potential problems).

“Tie software testing to business objectives.”

We help other people to mitigate the potential of losing something of value. Helping people in your company mitigate potential damage means giving the entire company the ability to progress toward strategic company goals at a sustainable pace, not at a reckless pace.

These goals are manifold (e.g., increase revenue, improve customer experience). We only indirectly influence these goals through testing but indirectly doesn't mean unimportant. With this oversimplified example, we just want to illustrate what we were trying to achieve.

“Unless your story isn't their story, they wont give it a chance.”

We were trying to find a trading zone with our internal stakeholders (e.g., managers). We did that by tying testing to their objectives. So, we made testing legible to them by adapting our testing terminology without oversimplifying the complex nature of testing too much.

In a nutshell, we made our story their story. The reason is simple: If our story isn't their story, they wouldn't give it a chance. All in all, we were trying to convey that we create value through testing by helping other people to create value.

Principle of Differentiation

Know where you end and others begin.

Our principle of differentiation addresses the second question we usually received from our management: "Why do we hire professional testers at all?".

Management tends to believe that anyone can test. Well, it's true: Anyone can test but not anyone can test well. In other words, management usually underestimates the difficulty of testing well. We addressed this question in the following way.

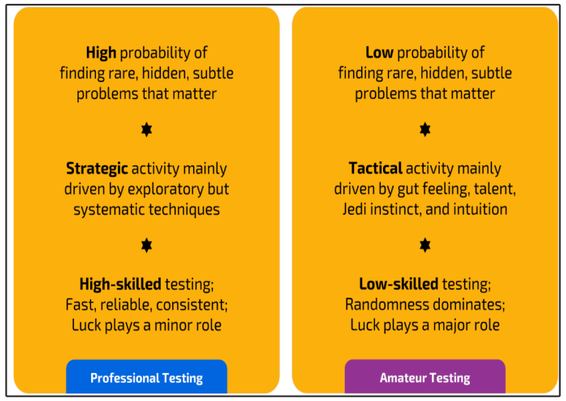
“Professional testing is a set of skills that must be learned.”

First, we distinguished between professional testing and amateur testing. Amateur testing is just code for testing that can be performed by anyone. Amateur testing has a low probability of finding rare, hidden, and subtle problems that matter. It's shallow testing.

On the other hand, professional testing has a high probability of finding rare, hidden, and subtle problems that matter. It's the ability to test deep. Deep professional testing leads to comprehensive product knowledge and risk coverage, shallow amateur testing doesn't.

Additionally, amateur testing is a tactical activity that is almost entirely driven by the gut feeling, talent, and intuition of amateur testers (e.g., developers, product owners).

In contrast, professional testing is a strategic activity that not only is driven by gut feeling, talent, and intuition. It's an exploratory enterprise backed by a colorful mix of systematic strategies, approaches, and techniques. For example, the ability to design and discuss test strategies is a hallmark of professional testing.



All in all, amateur testing is low-skilled testing. Here, randomness rules. Here, the act of finding problems that matter is like playing the lottery. Luck plays a major role.

In contrast, professional testing is high-skilled testing. It's a fast, reliable, and consistent testing activity where luck only plays a minor role in finding problems that matter.

Don't get me wrong. Shallow amateur testing isn't bad, far from it. It can be very important. Among other things, it makes deep professional testing possible (Michael Bolton).

How do I know? Well, I (amateur tester) have worked with Alice (professional tester). That's how I realized that I am not close but rather a world away from being a professional tester.

In summary, (excellent) testing makes people aware of problems (that matter), and problems that matter usually cause damage (e.g., loss of reputation).

This then begs the question: "Do you really want to rely on luck when your reputation is at stake?". We don't. We left management with these types of questions to motivate two things.

"Professional testing requires professional testers."

First, we emphasized the necessity and importance of professional testing to the business. Secondly, we highlighted that professional testing isn't natural talent only. It's a set of skills that must be learned. Simply put, professional testing matters, and since professional testing requires professional testers, professional testers matter too.

Here's a little exercise you can do. Think about your differentiators. Think about them in three ways. First, think about your unique differentiators. These are the capabilities that make you unique. These are the capabilities that only you possess as a professional tester. In business jargon, that's your unfair advantage. The ability to test deep is one example.

Secondly, think about your comparative differentiators. These are your capabilities that exceed the capabilities of other people (e.g., developers). These are the things you can do better. Typical examples are assessing risk, analyzing coverage, and telling the testing story.

Thirdly, think about your holistic differentiators. These are the capabilities that make you credible. For example, here you could highlight that you regularly speak at testing-related conferences, publish testing-related articles, or actively participate in testing communities to advance the testing craft. In short, think about where you begin and others end.

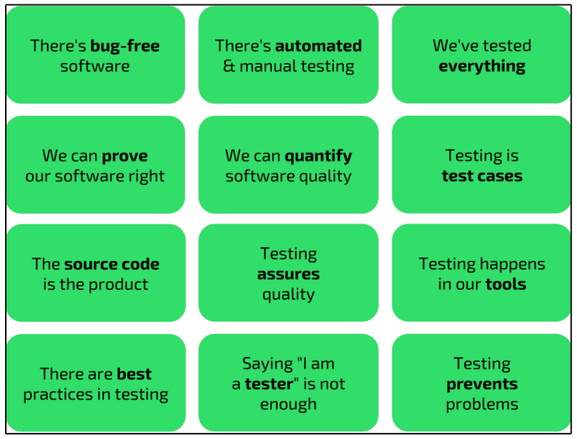
"If you don't understand your value, don't expect others to understand it."

It's important that these differentiators are true, important to your targets, and provable. Otherwise, it's just cheap talking. So, think about how you can defend them. In doing so, remember that less is more. The amount of information your management can absorb is usually limited. So, keep your list of differentiators concise but precise.

Principle of Debunking

Your success is built on preparation

Our principle of debunking is about anticipating, exposing, and proactively addressing toxic thoughts about testing. Think about all the things your management should stop buying, here and now. Make a list. Our non-exhaustive list is shown below.



For us, this list included, and still includes, the illusion of bug-free software, the fantasy that all testing can be automated, and the fallacy that 100% coverage is a meaningful practical concept. It included the popular misconception that we can verify software or that everyone, including cavemen, can test well. This list included the illusion that quality can be assured and quantified, and that testing is all about creating, automating, and executing test cases.

Whenever we pitched testing to management, we came prepared. We wanted to be quick on the trigger. We did that by having responses ready for these toxic thoughts. Remember, by failing to prepare, you are preparing to fail.

“Success tends to come to people who are prepared.”

For example, here is what we did to address the false belief that "testing is test cases". We reminded management that a recipe is not cooking. We reminded them that a sheet of music is not a musical performance. We also reminded them that a file of PowerPoint is not a conference talk. In the same way, a test case is not testing.

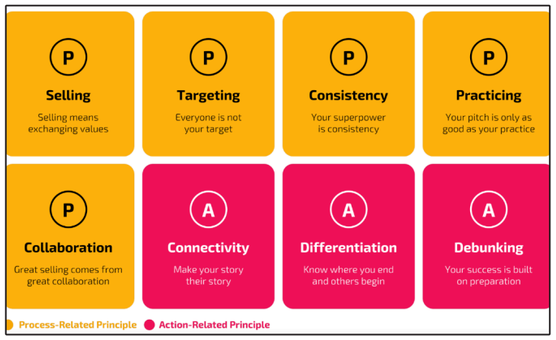
TA test case is an artifact, testing is a human performance. So, just as a file of PowerPoint is not central to a conference talk, a test case is not central to testing. Ergo, a test case is not a test. And likewise, the number of recipes (test cases) you have doesn't tell us anything about your cooking (testing) skills. This argument, among many others, has been stolen from the rich (Michael Bolton) and given to the poor (Ingo Philipp).

You get the point. We adapted our testing terminology and talked in simple terms to debunk these common misconceptions about testing. We used metaphors and analogies.

These are simple but powerful cognitive tools to compare software testing to something that's familiar to your managers. These tools helped us to clarify our idea about testing by turning the abstract discipline of testing into something concrete. It helped us to stop this spiral of toxic thoughts about testing. It helped us to separate myth from reality.

Conclusion

1 .Tactics without strategy is the noise before defeat.



The action-related principles were the general guidelines we followed in the act of selling testing, i.e., when we pitched the value of testing to other people (e.g., management). We also called them tactical principles.

The process-related principles were our strategic principles. They directed our processes that, in turn, helped us to plan, maintain, and scale the way we communicated the value of testing across the entire organization. This was crucial because the challenge was not to do that just for 1, 2, or 3 testers. The challenge was to do that for 40, 50, 60, or more testers.

I am highlighting this because I have seen too many testing teams that pay little to no attention to the process-related principles. I have seen too many teams over-focusing on action-related principles. This then often sets these teams up for failing in scaling.

Their action-related principles often turn into tiny little drops in a vast ocean. Ergo, act strategically, not only tactically, since tactics without strategy is the noise before defeat.

2 . Evangelism is not an option but a necessity.

Here are the three main benefits why following these principles pays off. First, think about what happens when a company restructures, the c-suite changes, or when employees must get laid off. In these situations, positions are questioned. Their relevance to the business is challenged. In the engineering department, people typically start questioning positions related to testing because it's tough to understand its value add. In these cases, you come prepared. You hit these questions hard and fast to make these concerns go off the table fast.

Secondly, through continuous and consistent evangelism centered around the value of testing, you probably won't receive these tough questions anymore. You already answered these questions since you never stopped addressing them. So, in these cases, it's already clear that professional testers significantly contribute to positive business outcomes.

Thirdly, think about what usually happens when new (agile) development teams are being formed. In the worst case, there's no discussion on whether professional testers should be part of these teams. Then there are situations where people discuss whether professional testers are required but decide against them. Again, through evangelizing the value of professional testing consistently and continuously, you'll see these discussions go away.

You will be included in these teams. People will understand that it's close to negligent to not include you. So, evangelism is not an option but a necessity to get your seat at the table.

3. You cant correct what you aren't willing to confront.

In hindsight, the journey of developing these principles was much more rewarding than the principles themselves. The real value of that experience was in all the things we've learned and unlearned in our heated discussions and lively debates.

The process of overcoming failures and finding new strategies in selling the value of testing is what is valuable. Through this process, we didn't just become better sellers of testing but also better doers of testing. Again, this takes effort but it's worth doing.

So, take the effort and accept the trouble that comes with it. The reason is simple: If you aren't taking care of how software testing is perceived by the people in your organization, others will. Remember, you cannot "correct" what you aren't willing to confront.

Namasté, my software testing friends.

“TESTER” IS AN OVERLOADED VARIABLE”

Recently I came across a post on LinkedIn explaining why there are no testers in Scrum[1]. What struck me most about the post was the amount of work the word “tester” was doing. In one sentence it meant one thing (a role in a team), in the next sentence something else (a step in the process), and so on. Hence the title of this post: the word “tester” was being used as an overloaded variable. So let’s do some unpacking.

Testers to people management

“Having a tester” means that there are people with the official title of “tester” or “QA engineer” or whatever within the company. For the purposes of people management[2], there’s a distinction between this role and the other roles in the company. This allows for more specific expectations about the role, for different career paths and salary scales, etc.

However, it tells you little about how what these testers do, is organized. There might be a separate reporting chain for testing all the way up the to the CTO or CIO. There might be a matrix structure, where people management is separated from work management. Or every member in the team reports to the same engineering manager or team lead. It also does not tell you if testers are part of the development teams[3] and if so, in what way exactly. The only thing it really tells you is that some people got a contract that lists for example “test engineer” as their job.

Testers as a process step

“Having a tester” means there is a testing phase after the development phase - implying a large and information-poor handover from development to testing[4]. As is often the case, the specifics matter here.

It is true there is always some testing you can only do after you believe all the code has been written. And if you have a tester, there’s usually[5] some handover to that tester. However, the timescale and the level of collaboration make all the difference here. A handover of 3 weeks of programming work via poor documentation is a very different beast from a handover of 3 hours of programming work via a short pairing session. The former type of handover leads to a world of pain, while the latter reaps the benefits that testers can provide[6].

Testers and their team

“Having a tester” means someone with the role of tester is involved in at least part of software development and delivery. This can take on two quite different forms: being a tester for the team versus being a tester of the team.

Tester for the team

“Having a tester for the team” means there is someone responsible for the work labeled as “testing”. If this tester is a member of the development team, they tend to be the sole owner of the “testing” column[7] on the team’s board. So in a very real sense, this team is operating as two teams: a development team and a testing team[8].

For this reason I think that this setup is in many ways the same as having a separate team of testers[9]. It’s not the team that’s doing the testing; testers are doing the testing for the team. If the tester is not available, little to no testing will happen.

Tester of the team

“Having a tester of the team” means that there is a team member who spends most of their time doing testing. All members of the team do all sorts of things, including testing. The tester of the team, however, has testing as their main focus, similar to how other team members have their own areas of focus.

Maaret Pyhäjärvi’s blog post “Tester roles and services” does a great job illustrating this way of working. She distinguishes 15 different testing hats and shows how each of the four team members either never wear that specific that, occasionally engage in that hat’s activity, or are focused on it. She re-posted the diagram on LinkedIn, saying: “It is possible to both have a tester in the team and have full team testing.”

So all of this to say what exactly? I guess nothing more than this: if you say that you have testers or that you don’t have testers, that does not tell me a lot. So I’d be curious to hear more about how you do software.

References and notes:

1. Which I don’t think is true. The 2020 Scrum Guide says “Scrum Teams are cross-functional, meaning the members have all the skills necessary to create value each Sprint.” and “Developers are the people in the Scrum Team that are committed to creating any aspect of a usable Increment each Sprint. The specific skills needed by the Developers are often broad and will vary with the domain of work.”

It does not say that every single Developer should have all the skills necessary. Nor does it not limit Scrum’s Developer-role to specific skills or activities. The Scrum Guide also states early on “We use the word “developers” in Scrum not to exclude, but to simplify. If you get value from Scrum, consider yourself included.”

2. I refuse to call it Human Resources. Emily Webber has a great blog post (and flowchart) on this topic: “Should you call people resources?”

3. This tends to be called “embedded testers”, which I find increasingly odd. It makes it sound you’re not a real team member, but someone inserted into the team from the outside.

4. What rarely gets mentioned in this context is the handover of bug reports from testing to development and the handover of fixes from development to testing.

5. Ensembling (mob programming/testing) is an excellent way to eliminate handovers.

6. The benefits a tester can provide, extend beyond the current topic of “checking stuff at the end”, btw.

7. It’s always interesting to see which kinds of testing do happen inside that column and which kinds don’t.

8. Two good definitions of teams: “If some of us can win, while others lose, we’re not a team.” (source?) and “A team is a group of people that share a problem.” (Douglas Squirrel)

9. They are not the same for activities that don’t fit in the “testing” column. A tester inside the team, even if it’s a for-the-team tester, will have more opportunities to participate in other team activities than a tester who’s in a separate testing team.

JOEP SCHUURKES

- Joep wandered into software testing in 2006. After a decade in which he learned (and practiced) exploratory testing and test automation, his focus shifted to a bigger question. How can teams and organizations build and deliver good software? To answer that question, he has been exploring topics such as technical leadership, agile coaching, and software methodologies.

Joep has given talks and workshops at conferences throughout Europe. He's also one of the organizers of the Friends of Good Software unconference and of the LLEWT peer conference.

~~Manual~~ Testing is Very Much Alive

Preamble

You may have noticed the strikeout in the article's title. It's not a mistake, but instead a recognition, perhaps a feeble one, that there's something wrong with the label of "manual testing". Years ago, it was simply called testing. There was no distinction between automation or not, but somewhere along the way the prefix got added. Consider this though: some surgeries are now done through robotics. Do we now say, "manual surgery?" Of course not, but why we say "manual" testing is beyond me. I'm not sure what the solution to this conundrum is, so for now, I'll just strike out the word to emphasize my point. I'll then stake a claim that ~~manual~~ testing is not only very much alive, but an often-overlooked critical skill when applied to static testing.

A Cause for Concern

We can't ignore the fact that test automation is here to stay, but for many, this could be a disruptive transformation. If you're a veteran like me, you've probably lived through several such disruptive transformations. This time though, I think things are different. For the sake of transparency, I still do ~~manual~~ testing, and I do not feel threatened by automation (in fact, I embrace it). For many readers though, i.e., those that devote much of their time to ~~manual~~ testing, you'd have every right to feel under siege, and I am here to tell you that I hear you. Many colleagues of mine have shared numerous stories about how their organizations are letting go of their manual testers and replacing them with developers who write test code, frequently called Software Developer Engineers in Test (SDET). We're slowly doing it where I work too, but to the best of my knowledge, these testers are moving into other positions within the company, and no testing jobs are being lost.

I maintain such bloodletting is totally unnecessary and wrong, and I'll spend the rest of this article revisiting and building on a concept known as Extreme Testing. In short, I believe my adaptation makes maximum use of the core testing skills, perhaps untapped, that effective ~~manual~~ testers already have. First, there's some underbrush to clear.

The Test Pyramid, Revisited

A common visualization of classifying tests is the classic test pyramid, such as this:

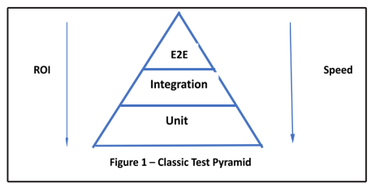


Figure 1 – Classic Test Pyramid

There's some controversy about this model, but for now, it serves its purpose. Structurally, I believe it's sending the right message, but as a pyramid, it's omitting something very important: a foundation! Let's give it one and discuss its ramifications.

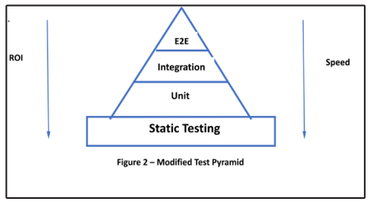


Figure 2 – Modified Test Pyramid

What I call static testing many people call reviews. On the surface, the subtle change in terminology might not seem like it's a big deal, but I believe it is. Reviews sounds too wishy-washy, whereas testing is a much more precise term. It also requires many of the core skills that make effective testers what they are, such as analytical and critical thinking, thinking outside the box, questioning everything, excellent communication skills, and more. By now, a light bulb might be turning on, but if not, I will give you a hint. For those familiar with the Context Driven School of Testing, they advocate, and I firmly believe, that testing is not checking. The former requires the skills I just mentioned. Checking, on the other hand, is usually repetitive, slow, and doesn't scale well to an agile initiative. It is a perfect candidate for test automation.

I know of several organizations that have recognized this and have effectively restructured their test teams, whereby testers write the test cases and leave the automation to the SDET's. In effect, testers are now writing specifications much like analysts. This is a step in the right direction. However, I believe there's a lot more that can be done to leverage the best and most demanding skills of testers. Before I dive into the details though, I want to expand on static testing and how it fits nicely into shift-left testing.

Shift-left Testing

I might be deviating from the standard definition, but I define testing as a process of exploration whose goal is to provide valuable information that helps to reduce risks and drive action. Add shift-left to the equation, and it means accomplishing this as soon as possible.

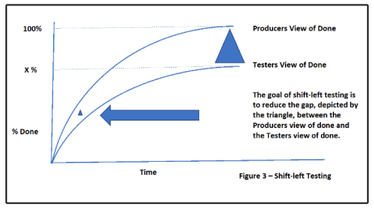


Figure 3 – Shift-left Testing

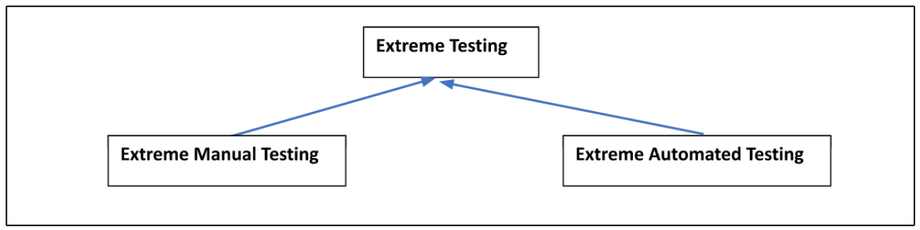
Together, static testing and shift-left testing have powerful ramifications. Moving the delta as far left as possible now exposes the true value of static testing, because we can leverage it to include not just user stories and the like, but everything that gets produced in a software development effort!

This may seem odd to someone who has been limited to testing code, whether it be scripted or exploratory testing, but true shift-left testing begins with artifacts that are produced well before requirements. For example, take the vision document. This is a perfect time to engage the test team for feedback. It's developed in the earliest stage of product development. As such, testers gain valuable insight into the intended product, its user community, and the problem(s) it is intended to address.

Extreme Testing

Many years ago, Extreme Programing (XP) placed programming at the hub of activity. A few years after its introduction, the 2nd edition of the seminal classic "The Art of Testing", 2nd edition, by Glenford Myers, et. all, was published, and in it, the authors introduced the concept of Extreme Testing. It built on the concept of automated unit testing introduced in XP, and added acceptance testing, albeit not automated. Certainly, unit and acceptance testing are necessary for an agile project, but why stop there? Since any deliverable can (and should) be static tested, let's incorporate it into Extreme Testing.

This now opens a whole new dimension on the scope of ~~manual~~ testing, and with it, the core skills testers possess.



Extreme ~~Manual~~ Testing (EMT) is multi-dimensional and can include static testing, exploratory testing, UAT, and more. EMT is not just a buzz word, but a true reflection that manual testing is as much as a first-class citizen as is programming and automated testing.

Exploratory testing has been thoroughly covered over the years, whereas static testing not as much, which is unfortunate because it is just as challenging. There's two parts to static testing. The easier part is testing the material that is there. The harder part is testing what's not there i.e., thinking outside the box and looking for what is missing. This requires orders of magnitude more skill, but it's also where testers truly begin to add significant value. Continuing with the product vision, a tester can identify feature gaps by learning more about the users, or perhaps doing research on competing products. The possibilities and opportunities are endless.

> This may seem odd to someone who has been limited to testing code, whether it be scripted or exploratory testing, but true shift-left testing begins with artifacts that are produced well before requirements.

A Simple Example of Static Testing

I happen to be a big fan of the classic triangle testing problem, first introduced in the 1st edition of "The Art of Software Testing" It goes like this:

The program reads three integer values from an input dialog. The three values represent the lengths of the sides of a triangle. The program displays a message that states whether the triangle is scalene, isosceles, or equilateral.

Myers goes on to give a definition of each type of triangle and the characteristics of their angles. He also has a complete chapter on inspections, walkthroughs, and reviews. Perhaps an oversight, he doesn't apply any of these techniques to his own exercise, and while countless articles have been written confirming or challenging his answer of 14 test cases, I've not seen anything written about static testing the specification. For example:

- While there are tests cases for unhappy paths, no mention is made about displaying an error message for them. I would consider this an omission in the specification.
- Is the input dialog a command line? A GUI? As a tester, this opens a host of related questions.
- There happens to be two definitions for an isosceles triangle: one has 2 sides equal, the other has at least two sides equal. While this may not affect your test cases, as a tester, I'd feel perfectly within my rights to do my own research.

Don't get me wrong though. My intent is not to criticize Myers and his groundbreaking book. Instead, it's to call out that no specification is perfect, nor is any oracle. For that matter, who defines what "perfect" means? Adding to the challenge, no software deliverable is perfect, and there's always the soft skills required to understanding unwritten context, or common knowledge. Just the same, this is exactly the kind of questions testers need to explore when they do static testing.

Looking Ahead

Behavior Driven Development (BDD) is gaining critical mass in man organizations, and for testers, this is great news, as they will be playing a vital role in the development of acceptance tests, usually called feature files. I hope to provide specifics in a future article.

Concluding Thoughts

Many practitioners and authors have made abstract statements like "you still need ~~manual~~ testing", but the specifics have always been lacking. I hope I have begun to change the conversation. ~~Manual~~ testing, and by extension, static testing can be applied to all deliverables throughout the entire software development lifecycle. My only hope is that you are given a supporting environment. If not, advocate for one. ~~Manual~~ testers, the world is now your oyster. Carpe diem!

Special thanks to Lalitkumar Bhamare for his review and feedback, which helped me crystalize and articulate my thoughts on manual testing.

References

1. Robot-assisted surgery - Wikipedia
2. "The Vision on the Future of Software Testing", International Software Testing Qualifications Board, December 2019.
3. "Just Say No to More End to End Tests", Mike Wacker, 2015.
4. "We Need to Talk About Testing", Daniel Terhorst-North, Tea Time With Testers, March 2021. TTwT_March_2021 (teatimewithtesters.com)
5. "Testing and Checking Refined", James Bach and Michael Bolton.
6. "Shift-left to make testing fast and reliable", Microsoft, 2021.
7. "The Art of Software Testing", 3rd edition, Myers, et. all, Wiley, 2012.
8. "Isosceles Triangle", Wolfram MathWorld.
9. Test oracle - Wikipedia

DAVID LEVITT

- Mr. Levitt is a passionate software engineer and educator. He's held lead roles as a programmer and tester and has advocated vigorously for allowing more time for static testing, both to his colleagues and his students. He holds a BS and MS in Computer Science and an Advanced Certificate in Software Engineering.

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

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 and to benefit next generation of testers with his work, we are re-starting his column.

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

Software Subcultures - Part 2

Pattern 0: Oblivious

We have added this pattern to the five used by other authors. Although it is not a professional pattern, it is the most frequent source of new programs, and can be used as a baseline against which other patterns can be compared. In pattern 0, there is no software development organization separate from the software user. An example of pattern 0 would be my developing a special little database to keep track of my own pulse and blood pressure, a spreadsheet to keep track of my scores at Precision Cribbage, or a BASIC program to drive a simulation game in one of my seminars. I have no manager, no customer, no specified processes. Indeed, I probably have little or no awareness that I am doing something called "software development," like Moliere's gentleman who was unaware that he had been speaking prose all his life.

If asked, I would probably say I was "solving a problem." That's why we call pattern 0, "oblivious."

Not only are the people using pattern 0 oblivious to their doing software development, but so are most writers on software development. I asked one of my clients, the Information Systems Manager of a large corporation, to survey the number of groups working in each of the various patterns.

Their estimates were:

0. Oblivious 25,000

1. Variable 300

2. Routine 2,600

3. Steering 250

4. Anticipating 0

5. Congruent 0

The Information Systems Manager told me he had never really thought about the 25,000 people in the organization who had been given access to PC's or time-sharing. He worried about what would happen when they became aware that they were doing software development. If they came to his organization for help, was that his job?

They would become aware, of course, only when their quality became unacceptable.

What saves Information Systems Managers from the Oblivious is a psychological phenomenon known as "cognitive dissonance." How many people will admit that they don't value the product of their own hands and brain? Indeed, this might be called the pattern of the "super-individual."

> What saves Information Systems Managers from the Oblivious is a psychological phenomenon known as "cognitive dissonance."

If asked why they are using this pattern, the Pattern 0 people would probably say, "nobody else can give me what I want, or really understand me." The characteristic magic posture of this pattern is that of a god: Omniscient and Omnipotent. At times, playing god can be a lot of fun.

Whether because of fun, cognitive dissonance, or some other factors, Pattern 0 is highly successful at producing satisfied users. Based on our casual observations, it seems to contain a number of sub-patterns. In this work, we are not particularly interested in Pattern 0 except as a standard against which other patterns are often weighed.

Pattern 1: Variable

Pattern 1, Variable, often follows Pattern 0 when problem solvers become aware, rightly or wrongly, that they are out of their depth. It is the first of the patterns to involve a distinction between the developer and user of software, so it's hard for the developer to remain oblivious to the process of software development. Because this is the first pattern to have this separation of responsibility for quality, it's the first pattern in which blaming appears as a substantial software development activity.

The super-programmer image

Crosby says of this pattern, "There is no comprehension of quality as a management tool," but we go a step further. A characteristic of Pattern 1 is that:

There is no comprehension of management as a development tool.

This pattern could well be called the pattern of the individual programmer. The ideal here is the "super-programmer," and the slogan is, "If we succeed, it's because of a super-programmer." A variant of this pattern has the slogan, "If we succeed, it's because of a super-team (led, of course, by a super-programmer). This is the idealized pattern for Mill's "chief programmer"—a compact "surgical team" headed by super-programmer. It is also the pattern described in hardware development by Tracy Kidder in The Soul of a New Machine.

When Pattern 1 is successful

Like all the patterns, this one is often successful. I commonly find this pattern in young companies producing software products for microcomputers. At the slightest provocation, any member of the organization will relate an elaborate "creation myth" about the heroic feats of the founding team. Often, as the new company grows, it evolves to Pattern 2, but retains the myths of Pattern 1. These myths have great value in recruiting new programmers. Thus, one of my clients spoke about the "small team" that worked on a project—later, I discovered that over 250 people took part at various times in its 3-year duration.

Another place where Pattern 1 is found to be successful is in a large organization where a "pool" of programmers serves some important group of specialists. Information centers are often structured as programming pools, but often they are more specialized. In aircraft companies, I have seen the pool attached to the engineers; in an insurance company, to the actuaries; in a bank, to the foreign exchange specialists. These pools can be highly effective at satisfying the needs of the specialists, and add much value to the company.

The ideal development structure

The ideal development structure in Pattern 1 is "the star in the closet." If the project is patently too large for one star, then the ideal is the "skunkworks." A Pattern 1 organization may have some procedures, but they don't cover most parts of the actual process. Besides, they always abandon any procedures at the first sign of crisis.

In Pattern 1, Curtis says, the typical personnel practices might include:

- Selection: Find out if candidate saw yesterday's game.
- Appraising Performance: Hold quick review before leaving on trip.
- Organization Development: Build morale over a beer after work.

According to Curtis, "software personnel are treated as a purchasable commodity," but I think the word "commodity" is imprecise. Personnel are "purchasable," but more in the sense that professional athletes are purchasable. The commodity model is more often seen in Pattern 2.

In Pattern 1, purchasing a "star" is the only hope the organization has of improving quality. The belief system is very much like voodoo (send in a hair or the fingernail of the key player, leader, programmer.) or cannibalism (which gives you the power of the person whose brain you eat.)

Humphrey says that the first step in statistical control is to achieve rudimentary predictability of schedules and costs. Since performance in Pattern 1 depends almost totally on individual efforts, the variability in schedules and costs depends almost totally on the variability in individuals. Studies of individuals have consistently shown variations of 20:1 or more in schedule, cost, and error performance among professional programmers, so it makes sense that this is the level of variation we see in Pattern 1.

In Pattern 1, the best predictor of project schedule, cost, or quality is which programmer does the job, thus reinforcing the belief system characteristic of this pattern. The programmer gets all the credit, as well as all the blame.

Pattern 2: Routine (but Unstable)

Pattern 2 arises for several reasons. An organization may be dissatisfied with the tremendous variation in Pattern 1. They may never have experienced Pattern 1, but simply need to build software that obviously requires more than a small team. Or, the projects may not be that big, but do require coordination with other organizations. In any case, managers decide they can no longer afford to "leave the programmers alone."

The super-leader image

Crosby characterizes the managers in this pattern as "recognizing that quality management may be of value, but not willing to provide money or time to make it all happen." There are several reasons they don't provide the money or time

- They don't appreciate the value of what can be accomplished.
- They don't know what is needed to accomplish changes.
- They believe that pushing the programmers is all they need to do the job.

A programmer in one Pattern 2 organization said of his management, "They think they're managing a salami factory." This pronouncement characterizes both the management style and the view of programmers who would prefer to be working in Pattern 1. The prevailing myth in Pattern 2 is that of the super-leader: "if we succeed, it's because of a super-manager (but there aren't very many of those). If we fail, it's because our manager is a turkey."

This attitude is expressed beautifully in the following excerpt from The Tao of Programming:

Why are the programmers nonproductive? Because their time is wasted in meetings.

Why are the programmers rebellious?

Because the management interferes too much. Why are the programmers resigning one by one? Because they are burnt out.

Having worked for poor management,

They no longer value their jobs.

Managers in this pattern do institute procedures—because they've been told that procedures are important to keep programmers under control. For instance, Curtis observes that by Pattern 2, management practices might have changed to:

- Selection: Managers are trained in selection interviewing.
- Appraising Performance: Managers trained in appraisal techniques.
- Organization Development: OD plan created, morale surveyed.

Both managers and programmers generally follow most such procedures. More

often than not, though, they follow them in name only, because they do not understand the reasoning behind them. That's why we call this pattern "Routine."

For instance, when Curtis observes that managers are be trained in appraisal techniques, that merely means there have been courses for managers. There is ordinarily no way to check on what processes managers actually use in their appraisals. When we do check, we find little correspondence between what the appraisal class outline said to do and what is actually done in appraisals.

When Pattern 1 is successful

Humphrey says that the Pattern 2 organization has achieved a stable process with a repeatable level of statistical control by initiating rigorous project management of commitments, costs, schedules, and changes. The operational word here, however, is "repeatable," not "repeated." A telling characteristic of the Pattern 2 organization is that they don't always do what they know how to do. Just when they seem to be doing well on a series of projects, along comes one "disaster" project that bypasses the procedures just when they are needed most. Worse than that, management starts taking action that further undermines the situation. Here's a memo issued by the manager of a project with a staff of 59:

We are now in the final push to bring Gateway to market. In the 10 weeks between now and turnover date, the following rules will be in effect:

1. Everyone will be on scheduled 10-hour days, 6 days a week. This is the minimum work week.

2. There will be no time off for any reason. All class attendance is cancelled. All vacation days are cancelled. Managers are not to grant sick leave days.

3. We must ship a quality product. It's everyone's responsibility to reduce the bug count. Testing, especially, must become more efficient. By ship date, today's bug count in every area will be cut in half.

4. We must ship an on-time product. Further schedule slips will not be allowed, and all previous slips must be made up by turnover. Starting today, any schedule problems will be reported to me on a daily basis.

Any developer, tester, or manager who violates any of these rules will be accountable to me. Remember:

WITH TEAMWORK, WE CAN FULFILL OUR COMMITMENTS.

You'll be interested to know that this product was shipped on time, and the manager was rewarded for his stunning feat of management. Some people did disobey orders, however, and got sick. Moreover, the "bugs" were not cut in half. Instead, they more than doubled, and four months after shipment, the product was suddenly withdrawn from the field.

Such disasters are inevitable in Pattern 2 organizations. Later, we'll use detailed models to demonstrate why. The primary reason, of course, is that Pattern 2 managers don't understand why they do what their routine procedures tell them to do. Thus, when things start to go wrong, they start issuing counterproductive orders—such as ordering people not to be sick.

The ideal development structure

It's a characteristic of Pattern 2 organizations to be desperately seeking a "silver bullet" to make a radical change in their performance. For instance, they often introduce refined measurements that make no sense in their unstable environment. Or, they purchase sophisticated tools which are either misused or lie on the shelf unused. This approach is what the anthropologists call "name magic." To work name magic, you just say the name of the thing: "structured programming," "CASE tool," "IBM"—and you have its full power at your disposal.

The ideal development structure for Pattern 2 is a manager supported by powerful tools and procedures. When the jobs are routine, all the manager has to do is ensure that everyone does every step in the right order. To do this requires "mana," the personal charismatic power that resides in an individual. If we just "put Jack in charge," everything will be all right. Unless it isn't.

Pattern 3: Steering

The competent manager

Pattern 3 managers never depend on magic, but on understanding. Though there are many exceptions, the average Pattern 3 manager is more skilled or experienced than the average Pattern 2 manager. Pattern 2 managers often have come a successful programming career with no particular talent for managing, no training in management, no great desire to manage, no time to acquire experience the job of management, and no role models to show them how to manage. That may be why they so often overestimate the power of their position:

I took a friend of mine—an organization consultant unaccustomed to working with such programming managers—on a consulting visit to one of my clients. After three days helping me interview people to determine the state of the organization, I asked him what he thought of their management style.

"Evidently," he said, "the only style they know is 'Management By Telling.'"

Pattern 3 managers have a variety of skills required to steer an organization, so they don't have to fall back on telling when their project gets in trouble.

When Pattern 3 is successful

Pattern 3 managers either have more training and experience, more desire, or else they are stamped from a different mold. Their procedures are not always completely defined, but they are always understood. Perhaps because of this understanding, Pattern 3managers generally follow the processes they have defined, even in a crisis. That's why they can successfully manage larger, riskier projects with a greater degree of success.

If you examine the "typical" project, Pattern 3 may not look spectacularly better

than Patterns 1 and 2. In Pattern 3, however, more projects are "typical," because there are many fewer outright failures. When a project starts, you can bet it will finish successfully—with value to the customers delivered on time and within budget.

The ideal development structure

Of course, Pattern 3 processes are more flexible, because managers choose them on the basis of their most recent information about what is actually happening. That's why we call this the "Steering" pattern.

Life in a Pattern 3 organization is much less routine than in a Pattern 2 organization, and the programmers are generally much happier with their work. They often display contempt for Pattern 1 programmers who don't appreciate the joys of working in a well- managed operation.

Humphrey says that the Pattern 3 organization has defined the process as a basis for consistent implementation and better understanding. He adds the important observation that advanced technology can usefully be introduced into this Pattern, but no earlier. In Pattern 3, tools are actually used, and used rather well.

To be continued...

PART 3 IN NEXT ISSUE

INTERVIEW

In this month's edition of TTwT, I have the pleasure of interviewing James Thomas.

Greetings, James! Before we get started, I'd like to take this opportunity to thank you for taking the time to share your perspectives to TTwT readers.

To set the stage, I'd like to dive deeper into your enlightening blog post on the Great Post Office Scandal.

Before I get started, it's only fair to mention that I downloaded the complete book but did not have enough time to finish it. However, I did get the chance to read some of it. It's an amazing story that touches on the intersection of so many dimensions, including, but not limited to testing, dysfunctional organizations, fear, ethics, and what might be controversial, a call for licensure. I'll try to cover each of these topics in the questions that follow.

The Great Post Office Scandal? What is it? What does it mean for testers? What good are certifications for? James Thomas over a cup of tea with Dave.

- INTERVIEWED BY DAVE LAVITT

Readers:

What makes this story so sad is that it's not just one of many massive IT project failures, but the year's long hardship it caused for many innocent people, some of whom went to jail or had their savings depleted due to no fault of their own.

To get the most out of this interview, do please take a few moments to read James's post. To further appreciate the story, I encourage you to visit the publishers website where you can read the 1st three chapters for free: The Great Post Office Scandal – Bath Publishing Limited. They are brief, and worth the few minutes of your time!

The book calls out there was a total breakdown of testing. The usual suspects have been identified, such as lack of adequate time, lack of auditing and accountability, etc. However, what I did not see mentioned in the book, and correct me if I am wrong, is that testing should have begun long before the first line of code was written, like in the initial planning or vision for the product, whose scale was clearly in uncharted territory. Of course, hindsight makes geniuses out of all of us, but imagine you were on this project in its early stages.

What techniques might you employ to raise awareness of the testing function?

I am glad you enjoyed the blog post, Dave, but I wouldn't regard myself as an expert on the Horizon case. I think I'll probably take any very detailed questions in a more general direction and refer you to James Christie for really deep and insightful analysis.

What techniques would I use to raise awareness of the testing function in the early stage of a massive project in a large organisation no doubt rife with politics? I guess I'd give different answers depending on the level of direct control or influence I had with other participants.

If I'm running the project then I'll bring in people capable of critical thinking and ask them to look holistically for potential risks, of what, to who, and when. If I'm not running things, then my advocacy for testing starts before the project, occurs in the project, and runs parallel to the project.

Before the project, on other projects, I'll work hard to make the value of testing visible to people who matter in the business. How? By doing the right kind of testing at the right kind of time and the right kind of cost, and then talking about it. I want people on the project to already be thinking that they'd like the benefits they've seen testing deliver elsewhere.

JAMES THOMAS
–

James Thomas is Vice President of the Association for Software Testing, a non-profit organisation dedicated to the advancement of the testing craft. Over the years he's had many roles including developer, technical author, technical support, and manager, but the combination of intellectual, practical, and social challenges in testing are what really excite him.

He's on Twitter as @qahiccupps and blogs at Hiccupps

By default I'll do all of those things as respectfully as I can, in relevant places, at appropriate times. That includes in meetings, in meeting notes, in tickets, in reports and white papers and so on. Again, the goal is to get people to see the value of testing to the project outcome.

In parallel I'll do the same kinds of activities with other people who are on the project, but one-to-one. I'll be building relationships, looking for allies and collaborators to help support a case for the critical review of the work being done, the way we're doing it, and the results we're expecting to deliver.

What I did not see mentioned in the book was the methodology that the project employed, but by virtue that it mentions lack of testing time, it hints at waterfall. However, I have seen, and have been a part of failed large-scale agile projects as well. I will cut to the chase. Methodology and technology are certainly important, but the three most important ingredients to a successful software project are PEOPLE, PEOPLE, PEOPLE. Extending further, the biggest driver is not the people in the trenches, but at the highest levels of management. For reasons that totally escape me, software has always been treated differently than other disciplines.

For example, I have been in organizations where the CIO has no formal training in computer science, software engineering, or has written any serious production code. Yet, this in unfathomable in any other profession, say accounting. I have long believed this dilemma would make for a fascinating investigation. I would be interested to know if you are aware of any such studies in this area?

I think my first answer reflects your point about people. The relationships between individuals, and their motivations, can have a massive effect on the success of projects.

I have never tried to look for literature on the specific question you asked but I'd say the fields of sociology, ethnography, and psychology would have direct relevance. I'm not sure they're more relevant to software than other industries where there is large-scale collaboration though.

For me, a CIO lacking technical expertise isn't as unfathomable as you suggest. C-level positions are more likely to be strategic than tactical and strategic thinking can be a transferable skill. That's not to say that subject matter and domain knowledge isn't valuable, of course.

For me, a good chief will have trusted lieutenants that can give them sufficient data, detail, and context. They'll also establish a network for triangulating that information and getting feedback and criticism on their thoughts and actions.

Getting back to the people in the trenches, the fact is that people in management positions are usually trained in negotiating and conflict management skills, but I can't honestly say the same for people in technology positions. Even worse, people in management often are many years older and have more experience and influence. I like the fact that your post mentions the ACM code of ethics. As an IEEE member, I am bound by the same code. However, the code by itself is no match for preparing practitioners for the power dynamics that exist in organizations. Do you have any thoughts on what is needed to provide this kind of training and guidance?

When I was managing a team of testers I organised annual team training, which I attended too. We had a pattern of technical skills one year and then (inter-)personal skills the next. One of the most productive that we did, for me at least, was on assertiveness.

If you're unfamiliar with the term, assertiveness might sound like aggression, but it's actually nothing of the sort. Assertive behaviour is professional behaviour: expressing yourself calmly and clearly and on equal terms with others, without backing down or being manipulative.

In particular, one simple conversational technique from that course has stuck with me for years: Express, Listen, and Field. It means say what you want to say, actively listen to the response, and then push parts of the response that are irrelevant to your point to one side. The course also gave us a vocabulary for talking about how we talked to each other, which was itself powerful.

When we introduced line management in my team I suggested to each of the new managers that they try to tease out a set of principles that they could use as a basis for making choices. I personally found that it invaluable for consistency of thought and action, and to avoid having to work every situation out from scratch.

A couple of other things I think it'd be good for people to be familiar with at work: the Satir model of interaction which describes how a message moves from inside my head to inside yours, and says that the likelihood of it being the same in both places is small; the perils and pitfalls of feedback and what it says about the giver and receiver; and congruence, the idea that decision-making should consider the self, others, and the context.

Much of this I learned from reading Jerry Weinberg.

I must believe there was at least one tester in the Horizon project who, in spirit, tried to abide by the code of ethics, but did not out of a real-world fear of losing their job. Consider a scenario where this individual had a family with financial obligations. It's easy to say, "walk with your feet", but it's not so easy to do in practice. What is needed, among other things, is a safety net so individuals can, and are encouraged to do the right thing. As I had not had time to completely read the book, was there any sort of a safety net at Horizon? if not, does England have a whistle blower act, such as the one in the US? See, for example: Whistleblower Protection Act - Wikipedia

There is government guidance on whistleblowing, yes. It includes this passage:

As an employer it is good practice to create an open, transparent and safe working environment where workers feel able to speak up. Although the law does not require employers to have a whistleblowing policy in place, the existence of a whistleblowing policy shows an employer's commitment to listen to the concerns of workers. By having clear policies and procedures for dealing with whistleblowing, an organisation demonstrates that it welcomes information being brought to the attention of management.

https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/415175/bis-15-200-whistleblowing-guidance-for-employers-and-code-of-practice.pdf

ISSUE 01/2022
OVER A CUP OF TEA WITH JAMES THOMAS

Here in the US, other than software engineering, all other forms of engineering require licensure, and rightfully so because of the risk to the public. Efforts at software licensure have been tried, and for many reasons have failed: Licensure of Electronic, Computer and Software Engineers (ieeeusa.org). Of course, licensure alone is not a silver bullet, but given the shortcomings of the code of ethics and the lack of adequate organizational training, it could certainly help. Assuming that one day there will be an appropriate exam for licensing software testers, and such licensure would be required for some people on some projects like Horizon, or on projects that are life-threatening, would you be in favor of supporting a licensing program or against it? Why or why not?

Big questions there, Dave, but I think your assumption is a big one too and "appropriate" is doing a lot of work!

First, I'm not against the idea of certification of practitioners in principle.

But what problem is your license trying to solve, for who? What would it license a tester to do, and on what basis?

You mentioned an exam, but there are other options too, including practical demonstration and assessment of skills and experience, bookwork, interview, or even simply (and sadly) attending courses.

Is it a cross-industry and technology license? Or is it tailored for particular applications with specialist concerns like AI or healthcare or the military?

In any case, why are we focusing on testers? Why not license designers, or developers, or product managers?

Why not certify the testing (rather than the testers), or maybe the products, or the companies who produce the products, or the regulatory frameworks in which companies operate?

And so on.

The Association for Software Testing has a well-considered position on certification.

The last sentence in the book is very sobering "As for any individual being formally censured or punished for their role in causing, perpetuating, or trying to cover-up the Great Post Office Scandal, well... I'm not going to hold my breath." I am going to go one step further and predict that most likely, those most responsible will not be held accountable, and something like this will likely happen again. Are you aware of any steps the British government has taken to ensure something like this doesn't happen again?

I am not. I'm also skeptical of the idea of ensuring things don't happen in general. That's particularly true in the complex systems (e.g. UK law) that exist to allow complex systems (e.g. the Post Office and Fujitsu) to build complex systems (e.g. Horizon) for use in complex systems (e.g. Post Office counters, their wide range of services and products, their partners, their suppliers, their staff, and customers).

Other than your blog post and what we've already covered, is there anything else about the scandal you'd like to share?

I mentioned complex systems a few times in the last answer. In its bare essentials the Horizon product is conceptually reasonably simple. It's a point-of-sales system which reports sales back to some central authority on a regular basis. But it was built and used by and in a web of complex systems with competing aims. It's not obvious that this complexity was appreciated or respected by the Post Office and Fujitsu, and that was compounded by a lack of empathy and consideration for the people involved.

For my final question, is there anything else you'd like to share other than the scandal? It could be related to any of your other posts or any special topics of interest.

I've recently been working on building clients of a service that use randomness and a model of the service to help me explore it. I've found it an extremely enjoyable and productive way to get the system into unexpected states, refine my understanding of the system, and generate data about the system that I can explore.

I did a talk on this for an Association for Software Testing webinar recently.

It's been a pleasure, James! Thanks again for your time. It's on my bucket list to visit the UK one of these days. Hopefully we'll get the chance to share a pint and talk more on these topics.

Cheers, I'd like that.

A couple of other things I think it'd be good for people to be familiar with at work:

The Satir model of interaction

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.

AUTOMATION WITH A HUMAN TOUCH

MARKUS GÄRTNER

–

Markus Gärtner works as Organizational Design Consultant, Certified Scrum Trainer (CST) and Agile Coach for it-agile GmbH, Hamburg, Germany. Markus is the author of ATDD by Example - A Practical Guide to Acceptance Test-Driven Development and contributes to the Softwerkskammer, the Germany Software Craftsmanship movement.

He blogs in English frequently at http://www.shino.de/blog.

It's been a while since I read from Taiichi Ohno about the Toyota Production System and from Goldratt about the Theory of Constraints. Thus far I thought, both have close to nothing to do with each other. Today, however, I got an insight that brought the two closer together for me. Let me explain.

Some context

I am currently working on a client that deals with 6- or 7-joint robots, even in the industrial field. Today I worked with one of their teams on their product vision. They had identified customers as companies that want to automate portions of their work stream by bringing in a robot and teaching it some repetitive tasks, as well as users in those companies, people who teach the robot the movements it should start to do later on while working on several workpieces.

While coming up with value propositions, I brought up something I learned from Taiichi Ohno while he was setting up the Toyota Production System in the early 1950s and 1960s: automation with a human touch. Ohno argues that his goal never was to automate every step of a process, but maybe to make use of humans with creative brains aiding in automation.

I brought up how the robot can easily free the human brain from repetitive tasks in the work process, thus freeing the human brain from drought tasks. Instead, a trained craftsperson can inspect the workpieces that the robot has been working on and identify additional steps to apply maybe manually to finish off the coarse work that the robot easily repeated.

Theory of Constraints

As reflected in one of the wastes stemming from Lean, waste of human effort, it dawned on me, how this principle basically is applying Theory of Constraints Thinking to the problem of automation. Let me explain.

In the Theory of Constraints, every production process is limited by a constraint, the slowest or most time-consuming step in the production facility. The whole system cannot produce faster products than the constraint currently allows. Goldratt identifies five focusing steps to improve the system:

1. identify the system's constraint(s): Identify the current constraint (the single part of the process that limits the rate at which the goal is achieved).

2. Decide how to exploit the system's constraint(s): Make quick improvements to the throughput of the constraint using existing resources (i.e., make the most of what you have).

3. subordinate everything else to the above decision: Review all other activities in the process to ensure that they are aligned with and truly support the needs of the constraint.

4. elevate the system's constraint(s): If the constraint still exists (i.e., it has not moved), consider what further actions can be taken to eliminate it from being the constraint. Normally, actions are continued at this step until the constraint has been "broken" (until it has moved somewhere else). In some cases, capital investment may be required.

5. Repeat the process: The Five Focusing Steps are a continuous improvement cycle. Therefore, once a constraint is resolved the next constraint should immediately be addressed. This step is a reminder to never become complacent – aggressively improve the current constraint...and then immediately move on to the next constraint.

If the constraint in a production process is the creativity of a human crafter, one way to exploit, subordinate, and elevate that constraint lies in freeing the human mind from boring, repetitive tasks that break his creativity zone.

Sources of explanation

So, if we bring in automation to free the human mind from the repetitive steps in that process, we make sure that the constraint of human creativity can be used to think about more problems.

Does this relate to test automation?

I think this point can be related to the benefits you can get from following a balanced approach between test automation and exploratory testing as well. Make sure to keep that human touch though. In software testing, usually, there is close to an infinite amount of tests that you could potentially run.

As Doug Hoffman explains in Exhausting your test options, he was tasked in his career once with testing the square root calculations of a new floating point unit. In order to make sure all input values resulted in the correct results for 32-bit floating point numbers as inputs, he pre-calculated on another device the expected results, automated all 4 billion calculations with the new calculation unit, and let it run on the new computer. Through the automation, the execution took 5 minutes. He found two erroneous results.

Thinking forward to 2022, we currently deal with 128-bit floating point numbers in more modern computers. Assuming the same execution times would yield 1^40 hours to calculate (several millennia). Thus, it's impractical to test all the values on a modern computer, and just consider how more complex than calculating square roots modern applications have become. In fact, the whole software testing theory mainly deals with ways to reduce the number of tests that you execute to some meaningful subset of all the tests you could potentially run.

As the story also illustrates, Doug Hoffman used automation to aid in his testing with a human touch. If automation takes away the boring tasks from the human brain cells, we can spend more time exploring what our automation did not catch. I came to realize how this as well relates to Theory of Constraints Thinking applied to software testing. And I began to realize how the similarities in other places of work have similar constraints.

Then it dawned on me, that the whole fear about automation taking away jobs from folks, is more an emotional one rather than a logical one. But maybe we need to dive into that distinguishment at some other point in time.

We are used to using different products regularly. There might be millions and billions of several applications but we are very specific when we choose to navigate through specific applications and websites, aren't we?

Have you ever found it questionable why a few are doing great while others are not?

The answer remains in the product design.

Let's think we have ideas for different products like online shopping, food delivery, etc. and we are building the product with the hope that it will create a buzz in the market and will be used by a huge number of customers.

There is a potential cause that might lead to an unsuccessful launch if it does not meet user expectations.

And this is why **Concept Testing** is coming into the picture and playing a significant role.

What is Concept Testing?

Concept testing is a research method of testing new or hypothetical products or services before they are launched that includes user feedback during the upfront part of the design process to give feedback on potential solutions.

It generally happens at the beginning phase of product development allowing users to share in the initial shaping of an idea to solve a problem.

The testing is specifically intended to screen several concepts to identify the strongest ones for progression, to improve/refine the base product or service proposition, and/or forecast their likely success.

Purpose

Concept testing is mainly used for go/no-go decisions for a new product. Below are some examples of situations in which this study finds its usage:

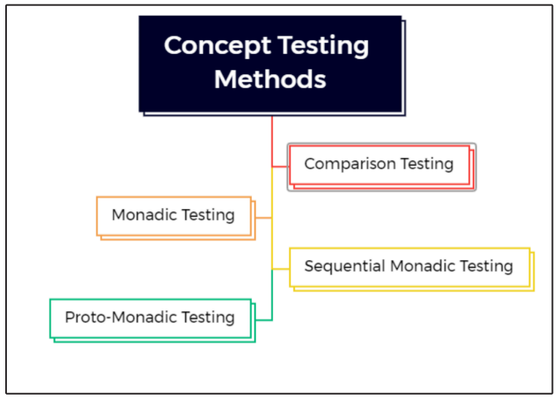
- Reaching out to a new market segment
- Ranking and selecting the best potential product concepts, brand name, packaging, logo, etc.
- Determining the optimal pricing point for new products
- Testing customers' trial experiences to see if the product or communications adjustments should be made.
- Benchmarking
- Forecasting demand

Involved Personas

The value in concept testing extends beyond UX and includes members beyond the UX team in the process of carrying out testing. This includes:

- Marketing team members,
- Tech/Dev team members,
- Business analysts,
- Key internal stakeholders from outside the core product team.

There are several types of methods for Concept testing, let's have a look.



Comparison Testing

1. Comparison testing involves the testing of two or more ideas.

2. Participants are given a **breakdown of multiple concepts together and then answer ranking questions to compare both concepts and choose one** that managed to sway them.

3. Researchers form inferences based on the respondents' answers, determining which concept proved to be more widely accepted.

This might result in little out of expectation as this is dependent on human preferences.

Monadic Testing

1. In a monadic test, the target audience is broken down into multiple groups. **Each group gets shown only one concept.**

2. These tests focus on analyzing a single concept in-depth.

3. Adds context to the respondents' choices. Results are much more thorough, giving researchers a better idea of what aspects, each group liked and disliked.

4. Monadic tests can be a relatively more costly endeavor, considering that it requires a large sample size.

Sequential Monadic Testing

1. Like the monadic test, in sequential monadic tests, the target audience is split into multiple groups. However, instead of showing one concept in isolation, **each group is presented with all the concepts.**

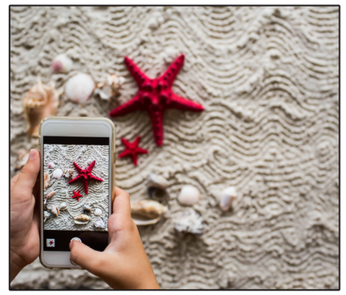
2. The respondents are asked the same set of follow-up questions for each of the concepts to get further insights.

3. Since each group of respondents sees all concepts, the target audience size required to perform a sequential monadic test is relatively small.

4. As each group is presented with all concepts, the questionnaires are relatively lengthy.

5. Multiple concepts can be tested in a single round. Thus, sequential monadic tests are **more cost-effective and easy to field. This concept testing method makes it ideal for research with budget constraints or when only a small target audience is available.**

Concept Testing



Proto-Monadic Testing

This is a combined idea altogether.

1. A comparison test follows a sequential monadic test, it's referred to as proto monadic testing.

2. Here, respondents first evaluate multiple concepts and are then asked to choose the concept they prefer.

3. The results of the latter play a significant role in verifying the results of the sequential monadic test. Whichever concept the participants prefer will likely have more positive feedback in the sequential nomadic test.

4. Researchers can verify if the concept selected in the comparison test is compatible with the insights collected about each concept.

Concept Testing Methods:

Hypothesis Testing

Hypothesis testing in statistics refers to analyzing an assumption about a population parameter. It is used to make an educated guess about an assumption using statistics.

Questions to explore the supposed requirement that the concept needs to address as a whole.

1. Do users struggle with accomplishing a task or have an unmet need in the area your concept will provide a solution?

2. Will this concept meet the needs of the users?

A/B Testing

A/B testing also known as split testing, refers to a randomized experimentation process wherein two or more versions of a variable (web page, page element, etc.) are shown to different segments of website visitors at the same time to determine which version leaves the maximum impact and drives business metrics.

This testing as defined helps us determine -

- Which version of the concept conveys an idea clearer?
- Which version of the concept provides a more immediate value?
- Which version of the concept is more realistic to bring to life and meet users' needs?

A/B testing requires greater resources in terms of creating an artifact to test. At least two versions of an idea are required to conduct A/B testing asynchronously, having users log into a site hosting visualizations of the concept and asking them questions along the way.

Interviews

Take interviews with stakeholders and end users to visualize their expectations better.

Current Comparative Products

Thorough research on current comparative products is highly recommended. This will set the stage for initial understanding and build a foresight regarding the market, and end- user surmise.

Benefits

Let's summarize the potential benefits that we get before launching the product -

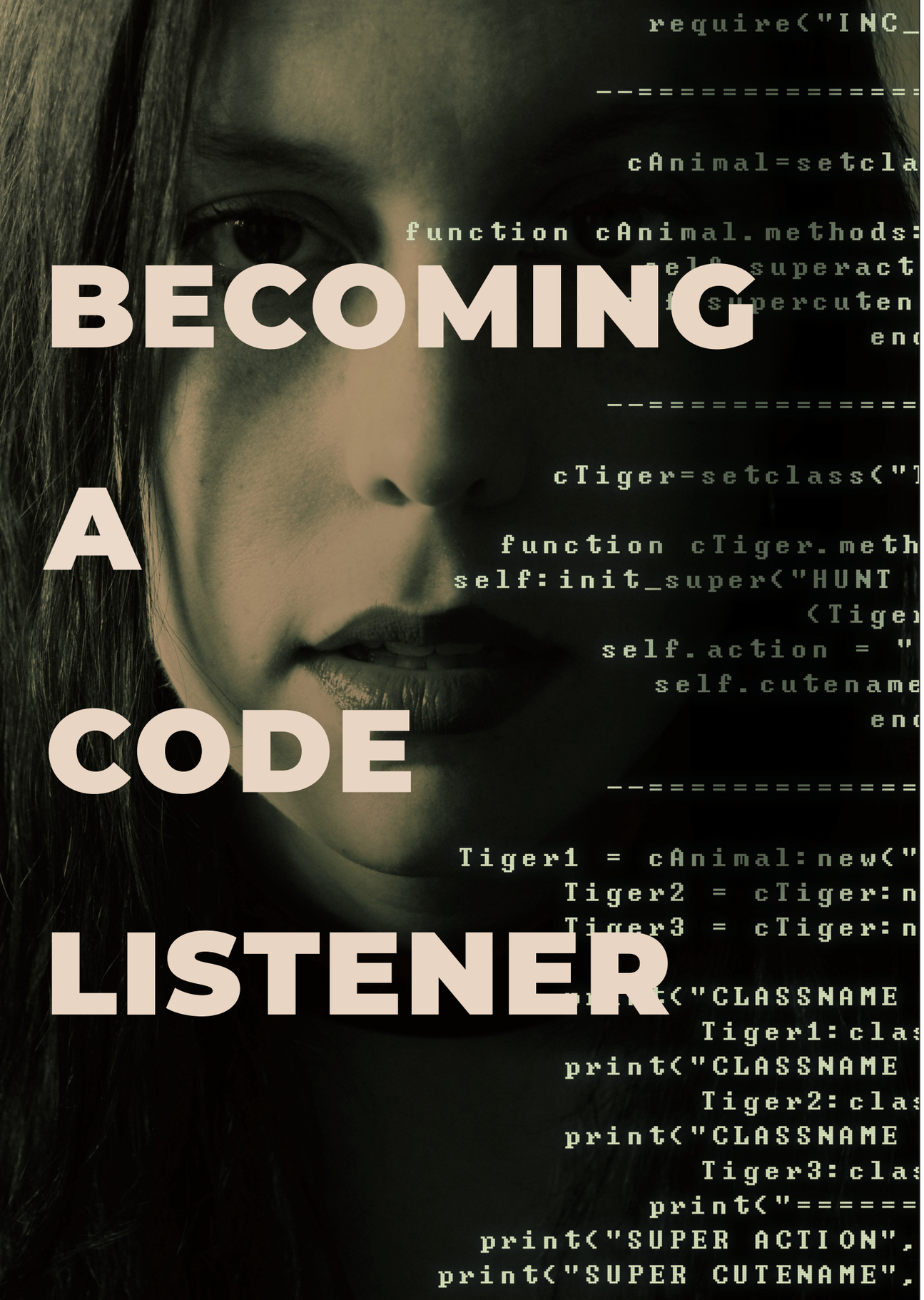
- Confidence that we are working on the right thing at the right time,
- Gain buy-in from stakeholders,
- Uncover features or enhancements you haven't considered.

An idea in your pocket? launch!

ANKITA BASU
–

Ankita Basu is currently working as an Analyst, a test enthusiast by heart. She likes cooking, being a plant mom, and reading self-development books especially. Ankita believes anyone can achieve his dream through hardwork and patience.

She loves to share her little knowledge and experiences through words. To find more about what this newbie blogger shares, check out Ankita Basu – Medium.



Introduction

Robert Sabourin and Mario Colina have worked in many different contexts in which testing professionals have been able to do great work, collaborating actively with other team members. Testers without programming skills can learn about technical risks through a process that the authors have labeled code listening.

In this article, we define the concept of “code listening” as the ability to identify test ideas from the source code of a software project. We define “test ideas” as any potential test objectives.

In this article, Robert Sabourin and Mario Colina will share several types of code listening illustrated with examples taken from real software development projects.

The methods described do not require programming skills and may be interesting to organizations, or development and testing professionals seeking to identify methods to improve their software testing process and practice.

Code listening skills are particularly effective when testers are working under time pressure and are interested in identifying areas of a product on which to focus their testing initiatives.

Understanding Algorithms and Program Logic

When tasked with a programming assignment, software engineers are challenged to solve a problem. The problem can be expressed as a requirement. The solution can be expressed as a design. The solution can be implemented with code and data.

Teams use a wide variety of styles to document software requirements and designs.

Formal, procedural methods such as the IBM/Rational Unified Process, RUP, use a modeling language to express requirements and designs. In RUP, requirements are often described using formal use cases. RUP represents software designs using various visualizations, such as sequence diagrams and state models.

Software development methods that rely on “agile” frameworks, such as Scrum, express requirements and designs with less formality. Our “agile” customers often express requirements with user stories with acceptance criteria and use various diagrams to visualize designs.

Active participation in planning activities provides an excellent opportunity for testers to learn about what can go wrong in a programming initiative.

1.1 Requirement Review

A requirement review takes place before a development team commits to implementing a feature. These reviews have different names depending on the software development lifecycle model. Many teams perform these reviews as part of “Refinement”, “Grooming”, “JAD (Joint Application Development)” or other “requirement elicitation activities”.

The authors encourage testers to actively participate in requirement reviews to help team members better understand what can go wrong, comparing new requirements with past implementation experiences.

Testers should be able to identify variables related to the upcoming requirement implementation. These variables can include factors or conditions that may influence or be influenced by the behavior of the software being developed.

1.2 Example of Code Listening Requirement Review “agile” Grooming

This is a short experience report by author, Robert Sabourin, about code listening related to requirement reviews in an “agile” grooming session.

Every Tuesday afternoon, the SCRUM team held a backlog grooming session. The team reviewed high-priority stories being considered for future sprints.

The team had four reasons to groom stories. First, team members attempted to gain a common understanding of upcoming work. Second, the team helped to clarify new story descriptions. Third, the team elicited meaningful examples to be used as acceptance tests, and fourth, the team estimated the size of each story.

Grooming meetings were time-boxed at 90 minutes. In attendance were the product owner, the business analyst, four developers, two testers, a technical writer, and the scrum master. The business analyst and the product owner authored the story descriptions. The business analyst also acted as the meeting facilitator.

The meeting started out with a short, five-minute team-building exercise designed to break the ice, get everyone talking, and encourage participation by all.

The business analyst used a PowerPoint presentation to walk through new stories one at a time. In this particular meeting, four stories were groomed.

One of the groomed stories was a new, complex business rule being added to existing transactions. The business analyst started by reading the story out loud, which was patterned to answer the questions: “Who is the user?” “What does the user want to achieve?” “Why does the user want to achieve it?” Testers asked plenty of domain-specific questions, uncovering potential ambiguity and risky misinterpretations. The team compared and contrasted the new story with existing functionalities. The business analyst edited the description as the team discussed clarifications. Some edits added acceptance tests in the form of examples describing the expected behavior in certain specific circumstances. Other edits clarified the statement of the story. The business analyst called on the authority of the product owner to confirm that the story was consistent with business needs. The team used planning poker to estimate the size of the clarified story. It took three rounds for the team to achieve consensus. During each round, team members justified low and high estimates using persuasive arguments based on recent relevant experience.

A couple of the new stories were very similar to previously implemented stories. The business analyst walked through them quickly. There was very little time spent discussing them. In both cases, the estimation required only one round of planning poker and all the team members agreed based on common shared experiences.

One story of interest was quite foreign to the team and raised many questions. It turned out that the business analyst and product owner did not have answers to some of the clarifying questions. During the meeting, the product owner and the business analyst unsuccessfully attempted to split up the story. The story was pulled from the grooming session. The product owner and business analyst would need to rework the story before subsequent team grooming.

At the end of the meeting, the business analyst summarized the work accomplished and indicated that the groomed stories would be added to the product backlog for some upcoming sprints.

1.3 Design Review

Design reviews take place during solution planning activities. In some, more formal, lifecycle models design activities adhere to project or corporate governance guided by software engineering professionals assigned architectural responsibilities. In some "agile" lifecycle models, design review takes place during a team's planning session.

Design decisions are taken by teams and will guide the implementation of any code changes required to implement the requirements.

Testing professionals can actively participate in design activities even if they do not have programming experience.

During design meetings, team members break down problems, starting from the abstract to the specific. This activity breaks down, or decomposes, a problem into technical tasks, which can be assigned to suitably skilled programmers. Teams will sometimes need to do some experimental programming to confirm that a design approach is both feasible and reasonable. Such experimentation is often called a "spike".

During planning meetings, teams make important design decisions related to:

Processing elements

- Which processing elements will be created, deleted, changed, or adapted?

Data elements

- Which data elements will be created, deleted, changed, or repurposed?

Algorithms

- How is the problem going to be solved?

The authors have seen testers actively participate in design sessions in three distinct ways: risk identification, design decision-making, and test oracle identification.

Risk identification can take place when testers correlate changes to processing elements with product usage models. Testers can identify excellent testing ideas, looking for unexpected consequences and side effects of changes to processing elements or data.

Testers can participate in design decision-making whenever a team chooses the best among several alternatives. As a heuristic guide, Robert Sabourin has frequently challenged developers to come up with at least three different possible solutions. The team chooses the best alternative by comparing the benefits and consequences of each solution. Testers can participate in design decisions by helping to identify risks associated with any one solution. Experienced testers can identify potential failures in each proposed solution.

When testers are aware of the methods being used to implement a solution, they can gain an understanding of test oracles, which are strategies used to assess the correctness of the implementation. The complexity of assessing correctness is often greater than the complexity of implementing a solution. When participating in design sessions, testers should be encouraged to identify different ways the implementation can be assessed. For example, is there a variable that must have values in a certain range or have specific relationships with other variables? Or should the computed tax be less than the cost of the items? Should a value be positive, or should an array be sparely populated?

1.4 Example of Code Listening Design Review System Decomposition

Decomposition of a software system

This is a short experience report by author, Mario Colina, about code listening related to a reviewing system design.

When testing large complex systems, how does one account for areas in the software system that are not so obvious to understand or test? When you're working on a large end-to-end system with interconnected external servers that have machine-to-machine communication with respect to each other, decomposing it down to solve a testing problem is one way to do it.

Take this scenario for example: a video recording and playback system over the internet, like online video streaming services. A video playback system has lots of interconnected parts with different components responsible for different functions like encoding/decoding, encryption/decryption, and file storage. Testing a playback feature is a daunting task and not so obvious with all the background processes running. Analyzing the software and hardware architectural designs to further understand the overall system can guide you in decomposing the system into smaller parts. These smaller parts of interest make it easier to test a particular function like how files are stored for video streaming.

For illustrative purposes, let's assume that the system stores and divides the video into many files that are 10 seconds each. This is called chunking. Let's also assume that the software component names the file in alphabetical order for the files that are spliced/chunked. The server would also create a manifest that would point to these chunked files once the whole file is processed.

The client device would request this manifest for playback. How can we verify that the proper files will be requested correctly during playback? If we observed a glitch or interruption during playback, how can we tell if this is due to a network problem or a software glitch during playback? One way to test that the chunking mechanism works is to

```
fileChunk001   2021-01-20 09:26:35.111000000
fileChunk002   2022-01-20 09:26:35.222000000
fileChunk003   2022-01-20 09:26:35.330000000
```

Another script can be written to scan the manifest file and verify that it is built correctly and contains the correct order of the filenames. Part of the analysis is to also verify false assumptions. If, for example, the system was to somehow splice one file out of order or a corrupted splice, then during playback, this would be evident for a file that is a 10-second chunk but will not be so evident for a chunked file less than a second during playback. This can be misinterpreted as a network glitch rather than a software component issue.

We have seen that drilling down to a small subsystem can help in testing one aspect of a larger system feature. Having good knowledge or asking about the architecture of the system helps in generating test ideas.

1.5 Code Walkthrough

When a software engineer has completed implementing code changes, the code is often subjected to peer review before being committed to a code repository.

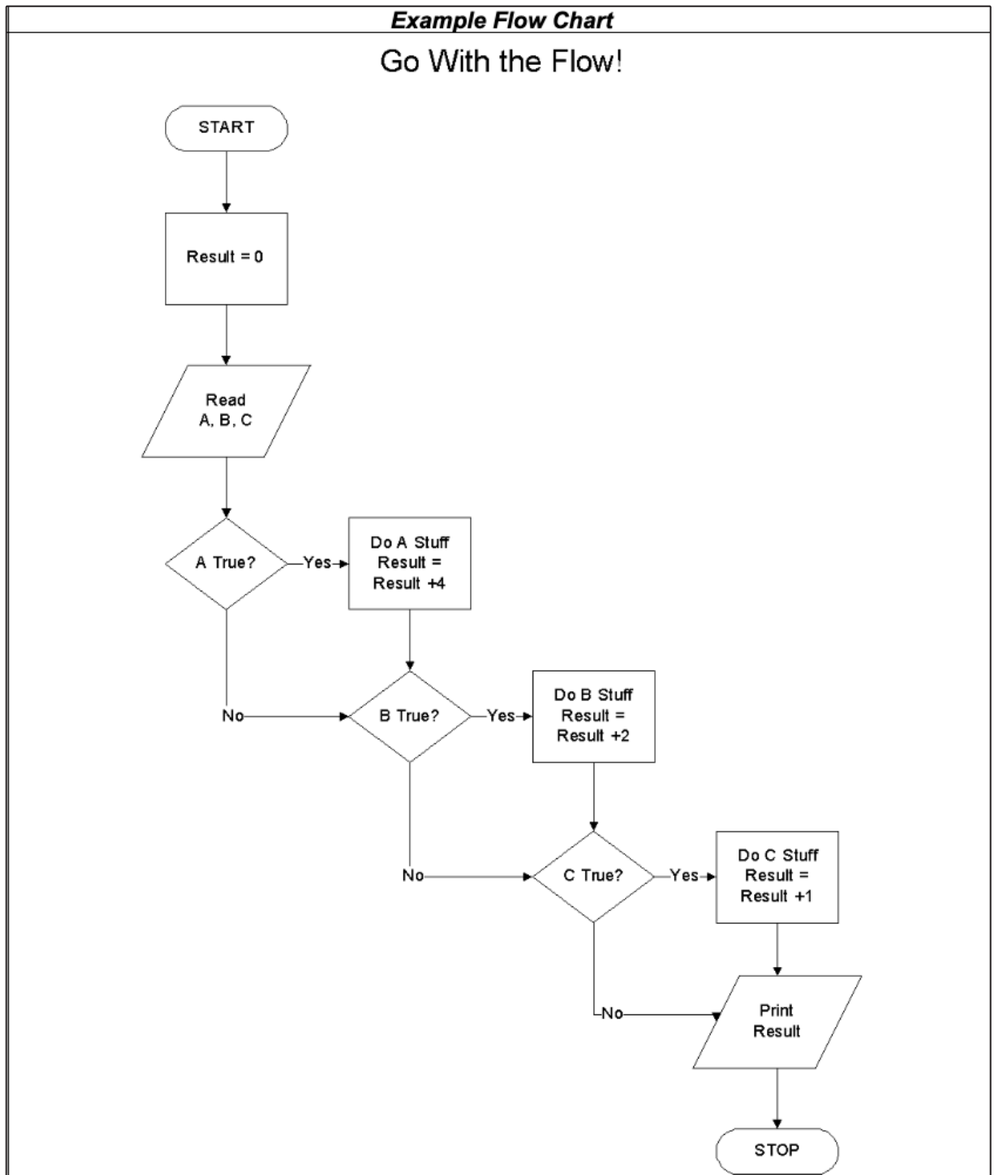
Testers with programming skills can actively participate in such peer reviews to verify that elements of the code developed match organizational standards and that the code fairly and accurately represents the designed solution.

Testers without programming skills can participate in a type of peer review known as a code walkthrough. In a code walkthrough, the programmer tells the story of all the codes that were written, or changed, in order to implement the design.

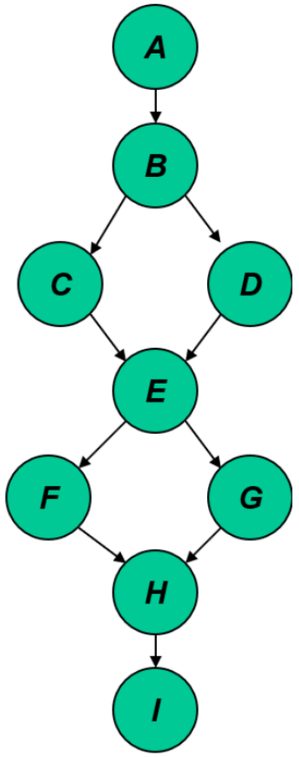
Testers can gain an understanding of the solution implemented, including the scope and complexity of changes to the product's code base, factors, and conditions that can influence the behavior of the changed code.

During code walkthrough, testers are encouraged to identify other factors that are missing from the code changes. A tester can determine if omissions are deliberate and not an erroneous algorithm or computation. A tester can ask programmers about the code that was omitted. Were omissions purposeful or were omissions a result of a design or programming error?

During the code walkthrough, testers can also create a visualization of the code changes to confirm their understanding of the technical work being done by the programmers. Control flow diagrams, decision tables, state models, and flow charts may be well suited to visualizing the code changes.



Example Control Flow Diagram



A control flow diagram is a visual representation of the pathways through a computer program.

Circles or nodes represent process steps.

Arrows or edges represent the sequence of steps.

Example Decision Table

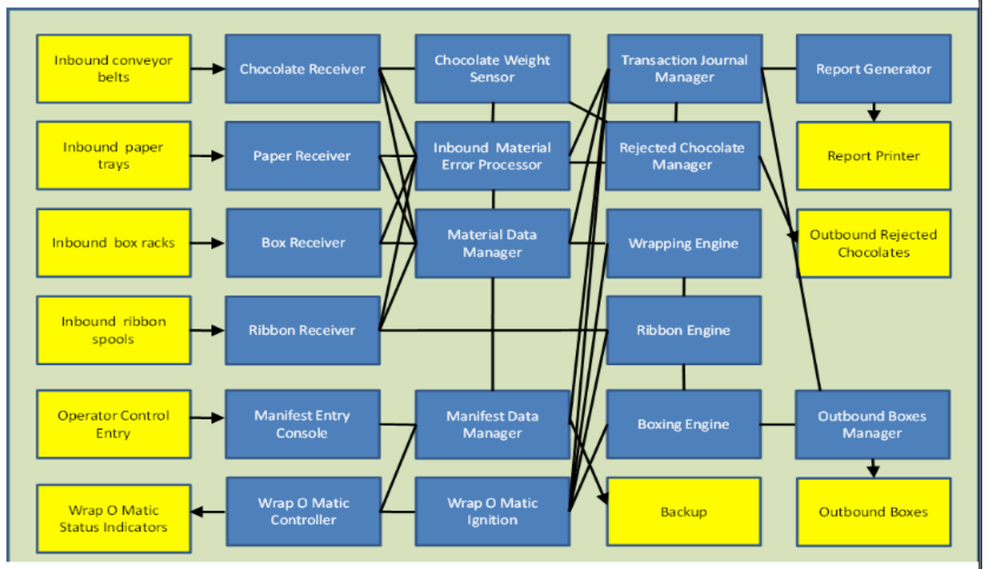
	R1	R2	R3	R4	R5	R6	R7	R8
CONDITIONS								
Roommate	-	-	-	-	T	T	F	F
Account has default	F	-	F	-	T	T	T	T
Pending non-default MVIN	F	T	F	T	F	F	F	F
Pending default MVIN	F	T	T	F	F	T	T	F
MVOT > MVIN	-	-	-	T	-	-	T	-
ACTIONS						-	-	-
Do not check "Default Move In" on MVOT	F	F	F	F	T	F	F	F
Adjust MVOT to match MVIN	F	F	F	T	F	F	T	F
Do nothing	T	F	F	F	F	T	F	F
Not valid condition	F	T	T	F	F	F	F	F
Check "Default Move In" on MVOT	F	F	F	F	F	F	F	T

A decision table is a visual representation of the logic implemented in a computer program.

Rows represent input conditions and output actions.

Columns represent the relationship between the conditions and actions, which are often called business rules.

Example Block Diagram

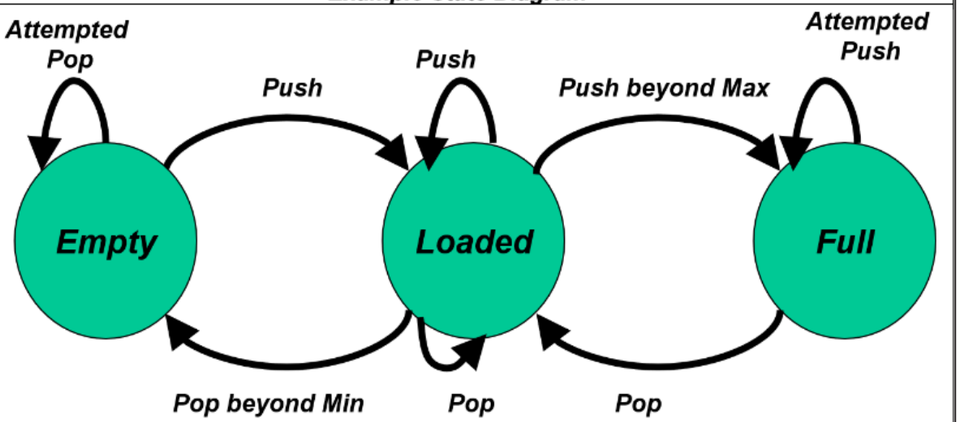


A block diagram is a visual representation of the structure of a computer program.

Blocks represent process elements.

Connectors represent the relationship between process elements.

Example State Diagram



A state diagram is a visual representation of the operating states of a computer program.

Circles or ellipses represent possible states.

According to Gerald M. Weinberg, “A state is a situation which can be recognized if it occurs again.”

Connectors represent transitions between states and indicate what event triggers the transaction and any associated variable output.

1.6 Example of code listening: A code walkthrough

Code Walkthrough

This is a short experience report by author, Mario Colina, about code listening related to a code walkthrough.

Understanding someone else's code is no cup of tea, especially if you're not a software developer. How else can a software tester understand the inner workings of instances, classes, and methods? Walking through the code with a developer can give you some insights into the thinking that went into the piece of software you're testing.

Follow me on this short journey. Many years ago, in a test environment not so far away, I was assigned a complicated feature to test. I had read the user story and the acceptance criteria. I knew what coverage I needed to test. I still wanted some more insights and to further understand how the complicated feature I was supposed to test worked. When the developer submitted his code for review, I thought to myself, "Hmm, I need to have a short walkthrough with the developer to show me the ins and outs of this feature."

I scheduled a meeting. While I sat with the developer, he explained to me how the feature worked. I was getting a clearer picture of the logic, the flow, and how the exceptions were being handled. This interactive session led me to develop more test ideas and think of ways I could have better coverage in my tests. I didn't need to fully understand the code, but I was able to follow the logic and the thought process the developer went through, and how he interpreted the feature requirements.

Meetings like these help to build a good relationship with a developer because it shows that as a tester, I care about understanding the feature beyond just reading an acceptance criterion. Having more of these types of meetings also allowed me to understand how to read and follow a programming language. This gave me the confidence to investigate and have a better understanding of the code when an exception occurred during my testing. It allowed me to provide more specific information in the bug report I raised and pinpoint the area in the code where the exception occurred.

1.7 Example of code listening: Understanding Algorithms and Program Logic

Code Listening for Program Variables.

This is a short experience report by author, Robert Sabourin, about code listening related to developing a device-independent graphics library application program interface.

I developed and tested several graphics libraries to be used in Computer-Aided Design systems.

One utility I worked on was programmed in three different languages. The device control software was written in C. Application-level software was written in Delphi and C++.

An important test objective was to confirm that an object's image was correctly rendered on the display. We knew the viewpoint and the object's position. Matrix calculations were used to map the three-dimensional object onto a two-dimensional projection, which was then rendered in a window. The window was a two-dimensional grid.

I reviewed the display code with its author. Several program variables were used in object rendering.

Object coordinates were stored as triples (oX,oY,oZ) using floating-point values.

The viewpoint was stored as a triple (vX,vY,vZ) using floating-point values.

The planar image was stored as a pair (pX,pY) using floating-point values.

An intermediate normalized coordinate was stored as a pair (nX, nY) using integer values.

The normalized coordinated mapped to a window pixel position was stored as (wX, wY) using integer values.

Mouse clicks were returned as window pixel positions. Reverse mapping computed the associated normalized coordinates, planar image coordinates, and object coordinates. One window pixel position could map to many possible object coordinates.

I learned that the range of values was for each data type. I defined tests selecting viewpoint and object coordinates so that all points of an object would render to the same point in (pX, PY), to the same point in (nX, nY), and to the same point in (wX, wY). I attempted the reverse mappings. I used domain analysis, boundary, and combination test design to exercise intermediate variables in all the different coordinate systems.

I discovered several image distortion and incorrect object selection bugs. Tests developed were subsequently used in regression testing.

Static Analysis

1.8 Static Analysis Tools and Continuous Integration

The authors use static analysis techniques to learn about product risks by studying non-executable artifacts of the software project.

Static analysis techniques fall neatly into two categories: manual static analysis and automated static analysis.

Manual static analysis techniques include reviews, walkthroughs, and inspections.

Automated static analysis techniques rely on tools to learn from project source code and other non-executable artifacts, such as database schemas.

In continuous integration systems, automated static analysis tools can be used to report risks based on the source code being built into a product. The static analysis tool would be run before or after compilation.

There are many commercial, community, and open-source static analysis tools available. Static analysis tools are highly dependent on the technology used to construct a software product.

1.9 Test ideas based on static analysis

Static analysis tools can provide a wealth of information about technical risks in the product being developed. Some static analysis tools focus on a very specific set of risks, while others are mostly general-purpose tools.

1.9.1 Code Metrics

Static analysis tools can provide information about the size and complexity of the product's source code. Size can be measured in many ways, such as counting the number of statements, counting the size of the compiled code, computing the number of pathways through a piece of code, and gathering other statistics about the code.

Variations in source code metrics may point to risky code changes. Large increases or decreases in code size or complexity suggest potential technical risks. A related test idea would be to exercise features that use code demonstrating variations in code metrics.

1.9.2 Code Standards Compliance

Some organizations implement coding standards and naming conventions to improve the reliability and maintainability of source code.

1.9.3 Application Program Interface

Static analysis of source code can confirm that any application program interfaces implemented are consistent with the way they are used. Static analysis tools can confirm that API parameters are of the correct type, identify missing parameters, identify parameters out of order when order matters, and also confirm that the API is in the correct operating state.

1.9.4 Potential Security Risks

Many static analysis tools are available to help teams identify security risks in a product's source code. These risks are highly dependent on the technology used to develop products. Security vulnerabilities flagged by static analysis suggest test ideas related to determining whether these vulnerabilities can indeed be exploited.

1.9.5 Maintainability

Some static analysis tools provide insights into possible difficulties in future code maintenance.

Code that is difficult to maintain can be easily damaged when programmers attempt to resolve bugs in legacy code working under time pressure. Tools can indicate whether there are too many pathways through a section of code, whether the code is commented on, and whether the code logic is confusing.

1.9.6Unreachable code

Static analysis tools can find sections of the source code that could never be executed. This unreachable code is often a side effect of cut-and-paste programming. Unreachable code can be an indicator of incorrect code reuse, logic errors, or missing conditions.

1.9.7 Risky patterns

Some static analysis tools can identify risky source code patterns particularly related to memory management or thread-safe coding practices.

1.9.8 Code Structure

A series of static analysis tools are available and can be used to scrape through a code repository and document the code structuring. The structure includes calling order, a summary of methods, and classes, including parameters and data types.

1.10 Example of code listening: Static Analysis Variance

Code Listening for Static Analysis Variance

This is a short experience report by author, Robert Sabourin, about code listening with a static analysis tool in a computer desktop security project.

I managed a software engineering team responsible for a home security software suite deployed to consumers through internet service providers on a subscription basis. Frequent updates were deployed to clients in response to new and emerging threats, computer viruses, spyware, trojan horse security risks, and other harmful malware. The software ran on MS-Windows computers. The code was primarily written in Microsoft Visual C++. The development teams used an iterative incremental Scrum-like framework. Independent testers were assigned to each project team. Testers collaborated with programmers continuously through each iteration. Testers had access to the source code and build systems.

Continuous integration included frequent builds with multiple tens of thousands of automated regression unit tests.

When ready (programming done, passed unit testing, and passed peer review), the build candidate was exercised by an independent tester.

Testers considered risks to explore, and over and above confirming requirements were acceptably met on target platforms. The tester reviewed code changes by studying some static analysis findings generated during the product's build process.

Homemade static analyzers, written in Perl, were used to assess (a) the number of statements in the source file and (b) estimated cyclomatic complexity by counting the number of basis paths in each source file. The tester would review changes in value from a prior build measured prior to the implementation of new features.

Whenever a significant variance (over 20% increase or decrease) was observed in statement count or complexity, then additional exploratory testing would be done on the features, capabilities, or workflows associated with the changed code. This heuristic helped testers uncover many important bugs that would otherwise have been missed by the regression testing (checks) done.

The tester code-listened for static attribute variance. The tester successfully identified features and workflows at risk of being adversely impacted by code changes.

1.11 Example of code listening, static analysis code structure

Javadoc

This is a short experience report by author, Mario Colina, about code listening related to visualizing code structure using static analysis.

Looking at code is not the only source to understand code. An immense aid in the understanding of what methods, classes, or functions do is to look at API documentation generated from code document generators. One such document is the Javadoc, which is a document generated from java code that documents classes. It is helpful in understanding the code because the method and classes are explained with all the parameters required for its use.

Let's continue the journey where, in this instance no pun intended, the application under test did not yet have a REST interface implemented. We used a test tool that allowed us to make method calls via an ejb interface (Enterprise JavaBeans).

Looking at the Javadoc was helpful in not only understanding the code but also in the creation of my test scripts. You might ask, "How?" Well, when writing my automation test scripts, I referred to the document for a better understanding of how the specific java method worked and what input parameters, such as strings, lists, objects, etc., were needed to make a proper method call to the application. It identified what were the mandatory or optional parameters that were used and the return values of the method. It also allowed me to test some invalid inputs and verify how the system would react.

Another good use of the Javadoc was when a java stack trace exception occurred. I cross-referenced the method from the trace to the Javadoc to get a better understanding of what that method does, obtaining more information in my investigation of the exception.

The document is a great source of reference when it comes to troubleshooting the application under test in conjunction with our test scripts. Since the Javadoc (for this example) is an HTML-based generateddocument, it makes it easy to follow links to other methods or classes, and it provides a rich source of information.

Stack Trace

1.12 Program Model

The authors encourage all testers to have a model of how modern digital computing systems behave. The computing model you use does not need to be complex but should include considerations of how computer resources are managed to execute the software we are called upon to test.

The basic model used in almost all digital computing systems was first described by John von Neumann. Von Neumann model includes the following six elements:

- Processing unit
- Control unit
- Memory for both data and instructions
- External storage
- Input mechanism
- Output mechanism

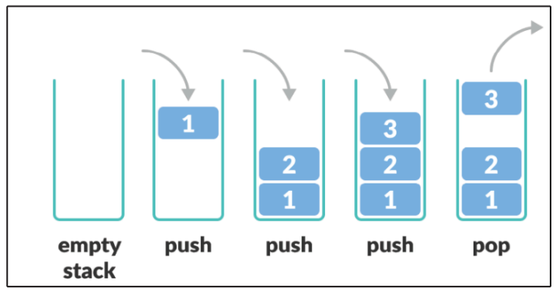
Since instructions are stored in the same memory as data, they are treated as data by the control unit. The control unit always has a pointer to where the current instruction resides in the memory.

1.13 The Control Stack

Software systems are composed of many modules, which call on each other to perform operations and computations. Control units keep track of the instruction used to call on a module. The structure used to keep track of the return instruction location is called a stack.

When a new module is called, the return address is placed on the stack.

When a module has completed its work, the control unit returns to the return address.



Basic stack operations include placing an item on the stack, pushing it, removing an item from the top of the stack, and popping it.

1.14 Calling Structure

The calling structure of the code that is executed is evidenced by the state of the stack.

When a programming error leads to a system crash; for example, trying an illegal operation, trying an operation with invalid data types, moving to an instruction outside of accessible memory, then a stack trace can be created by the operating environment (for example, the Java Runtime Environment) and this trace can be available to testers and programmers to determine the code pathway taken, which led to this unexpected exception.

1.15 Example of code listening: Stack Trace

Stack trace Review

This is a short experience report by author, Mario Colina, about code listening with a stack trace or a rest API middleware system.

Trying to decipher someone else's code is no walk in the park, especially when you're not a software developer. How else can a software tester understand the inner workings of instances, classes, and methods? Oh my!!! Should one wait for the system to crash and then investigate? Well.... Not exactly. A stack trace is useful for gaining insights into the inner workings of an application.

Are you getting tired of following me on this short journey? Hopefully not! This is fun! I was testing a middleware application written in java by executing REST API calls. Any time I am testing an application, I always follow my server logs in real time and watch out for Java Exceptions! Cry havoc!! Why did the application throw an exception? Was there not a handler for it? Stack trace logs provide me with vital clues of where the exception occurred, as it is a trail of breadcrumbs that leads you to the offending method. From the Sample trace below, the most recent method (methodTwo) points to the root cause of the exception.

```
xception in thread "main" java.lang.RuntimeException: A test sample exception
 at com.codelistner.stacktrace.StackTraceExample.methodTwo(AStackTraceExample.java:25)
 at com. codelistner.stacktrace.StackTraceExample.methodOne(AStackTraceExample.java:15)
 at com. codelistner.stacktrace.StackTraceExample.main(AStackTraceExample.java:7)
```

Further confirmation was needed. It was time to bring the developer in and show what I found during my testing. Our developer connected to my test system through the Integrated Development Environment (IDE). I would step through the actions from my test script one step at a time while the developer was following the path in the code until the exception occurred. We were able to isolate and confirm that methodTwo was the root cause. In some instances, the most recent method in the stack trace can be a symptom of another issue. But in this instance, methodTwo turned out to be the cause. In any case, the trace helped isolate where to investigate the issue.

Stepping through the code with the developer allowed me to have a better understanding of how the application worked and in what instances the application crashed. This provided me with new test ideas and new insight into what is happening under the hood, so to speak.

Another great benefit was to help build a better relationship with the developer. Stack traces are important for investigating issues and they help you to get a better understanding of the code you're not familiar with.

Code Coverage Tools

1.16 Coverage Models

Some software engineers, and testing professionals, believe that testing completeness can be measured using various coverage models. A coverage model can be defined as a means of measuring which elements of a software system have been executed, or otherwise used, during software testing activities. There are well over 100 documented coverage models. A testing activity may look to have a high coverage using one model but could have a very low coverage from the perspective of other models. A classic example would be that even if 100% of the source code in a model has been exercised, it is possible that many pathways through the module have not been exercised by the same set of tests.

1.17 Coverage-based Test Ideas

Test coverage tools enable testers to identify which parts of a software system have not been exercised during a testing activity. Coverage-based test ideas exercise elements of source code that were not otherwise exercised during a past testing activity.

1.18 Example of code listening: Code Coverage

Code Coverage Tools

This is a short experience report by author, Mario Colina, about code listening using code coverage tools.

Statements, branches, and decisions. Did we cover everything yet? No, we are talking about Code Coverage tools that can help testers identify areas in a code that have not been executed as well as areas that need to be tested with additional tests.

Alright, this is the last of my journey in that test environment not so far away. I was asked by the development team if I could help with some testing. Along with Javadoc, IDEs, and stack traces mentioned in the previous sections, another source of understanding the code is to understand how it navigates and flows in its internal network of "roads".

Metrics that a tool can collect are function, statement, branches, conditions, and line coverage. These metrics determine what tests are missing. In this oversimplified example, I was investigating branch coverage containing if/else statements.

```
read x

read y

z=x+y

if (z>10) {

  // code in if block executes

}

else {

  // code in else block executes

}
```

The code tool identified a gap in one of the branches not being executed—the else block—since the test values of x and y never added up to greater than 10. What the coverage tool also allowed was to understand the flow of the code. Understanding the "internal roadways" provides a deeper understanding of the product and gives rise to creative test ideas. I was able to ask questions like:

"How can I get to a certain block in the code from an initial starting point and what test data can I use to influence the path?"

"What data am I missing?"

"What conditions did I omit in my test design?"

Using the coverage tool provided me with more knowledge, which helped me to read codes better and understand the flow of branching. The more tools a tester can use, the better the building blocks to design tests for different test phases like feature, function, and end-2-end integration testing. It is another arsenal that can be used to gain more information and knowledge about the code.

Troubleshooting and Debugging

1.19 Troubleshooting

A lot of training focus testers on the act of identifying and reporting bugs in a piece of software. Rich terminology and vocabulary are established to emphasize the difference between errors, failures, issues, incidents, and defects.

The authors of this paper come from an engineering background. The authors consider it critical for testers to provide insights into the cause of a problem, not just the externally observed behavior of the system.

The authors define troubleshooting as a term used to describe the isolation of the cause of a specific problem.

1.20 Debugging

The authors define debugging as a series of technical activities directed at finding ways to resolve a problem.

In the book, "Systematic Software Testing", SST, author Rick Craig indicates that testing and debugging should be isolated from each other. Testers are expected to identify defects but are not expected to isolate their cause nor find a solution. Testing is not debugging. Testing is not troubleshooting.

1.21 Active participation by testers in troubleshooting and debugging

A code listening tester can work with a programmer in the development environment to identify factors that cause a problem and verify that a proposed solution yields the correct results.

Testers can pair with programmers and other team members to go beyond just identifying a problem; they can collaborate to help resolve the problem as well. You don't have to be a programmer to learn bug isolation and debugging techniques with a modern development environment.

1.22 Example of code listening: Troubleshooting and Debugging

Troubleshooting

This is a short experience report by author, Robert Sabourin, about code listening used in troubleshooting and debugging a complex insurance application.

Fred, Bill, and Chris are members of a scrum software development team at a major American insurance company. Their products compute insurance premiums based on a proprietary risk model. Fred is a seasoned insurance software tester. Bill is a senior programmer. Chris is an insurance domain subject matter expert.

Some projects are customer-facing projects, while others are exclusively for internal use. Newly developed products have multi-tier, web-based architectures. Legacy products have mainframe data-centric architectures.

Fred, Bill, and Chris often collaborate in order to troubleshoot bugs found in the heat of the sprint. They apply a wide variety of skills. Fred contributes with his scripting and test design skills. Bill contributes with his programming, debugging, and unit testing skills. Chris contributes with his practical business analysis skills.

Fred, Bill, and Chris troubleshoot problems in a workspace called the aquarium. They use a test environment running the previous product release with recent, sanitized production data. In parallel, but on independent hardware, they test a current development build. They simultaneously exercise the same business rules on both systems.

The test system has an integrated source-level debugger. Bill was able to insert breakpoints at any place in the code.

The test environment enabled the creation of custom test data and test transactions. Fred could simultaneously execute the same transaction side by side on the old and new systems.

Fred and Chris figured out different test oracles. Fred sought out at least three independent means of assessing correctness for tested objects, features, or transaction types.

During the early project sprints, Fred, Bill, and Chris ran into many unexpected difficulties. They found that transactions often gave different results in the old and new systems. Differences in behavior were not always indicators of issues requiring resolution. Some differences were related to variations in precision in intermediate calculations with new technologies. For example, COBOL was used to implement business rules in the old system, whereas Java was used to implement business rules in the new system. COBOL and Java data types had different numeric precisions.

Some differences were related to the order of operation. Computations were implemented differently in the old and new systems. Some computations on the old system were computed in their entirety before being stored in the database. These same computations may have been implemented incrementally, with intermediate results stored in the database rather than the entire computation.

Some differences were related to bugs in the old versions, which were being corrected in the new system. An old calculation may have been incorrectly programmed, being used, as is, for many years without the knowledge of the system users.

Some differences were due to Phantom Bugs, which occur when part of the code required to implement a transaction was not ready in the new system being developed.

Some differences were due to errors in the implementation of test transactions.

Some differences were due to bugs in the test harness rather than bugs in the system being tested.

And some differences were actually related to bugs in the new system being developed.

One of the concerns was that a bug in the old system could have been cloned in the new system, leading to a passing test. Two wrongs do not make a right. These errors could not have been found with the test strategy employed by Bill and Fred.

Bill and Fred worked with the business analyst to help identify acceptable variations on ranges of values. Essentially, for some transactions, a variation could be acceptable as long as it is within some rules. For other transactions, an exact matchup to an established precision was required.

Fred and Bill worked together to study differences in results in the test harness. Whenever a bug in the system under test was suspected, they used two-pronged combined bug isolation and debugging approaches in order to troubleshoot the problem. They worked top-down by building new test data sets, trying to isolate which variables trigger the error. Each test data set varied by a single factor and together, they were used to explore a hypothesis related to identifying factors influencing the buggy behavior. They worked bottom-up by placing strategic break-points, assertions, and data logs. Break points would stop execution at certain points in the code so that Fred and Bill could look at the program variables and, if required, single-step through the code to see if the code flow matches what was expected or what was intended. Assertions in the code were special pieces of code added to confirm that the system or a transaction was in the correct state at the time the code was running. For example, a file should be open before its data is used; if not, an assertion would be tripped, indicating that the code was executed in a way that was out of control. Data logs could be used to record in a separate log file or database any interesting intermediate variable. Fred and Bill could study the log after running a data set looking for inexplicable behavior patterns.

Sometimes, Fred and Bill used an approach called OFAT. In OFAT, each test data set would vary by one single factor at a time. OFAT was especially useful to help confirm or refute hypotheses that a small set of factors are culprits, but they do not affect the behavior in combinations. They are assumed to be independent factors.

Sometimes, Fred and Bill used an approach called MFAT. In MFAT, each test data set would vary by many factors at the same time. MFAT was especially useful to help confirm or refute hypotheses that a few dependent variables were causing the problem being investigated. Note that various combinations of test design approaches, including Pareto Analysis (80/20 rule), Randomizing Data, and Pairwise combination approaches, were used with MFAT troubleshooting.

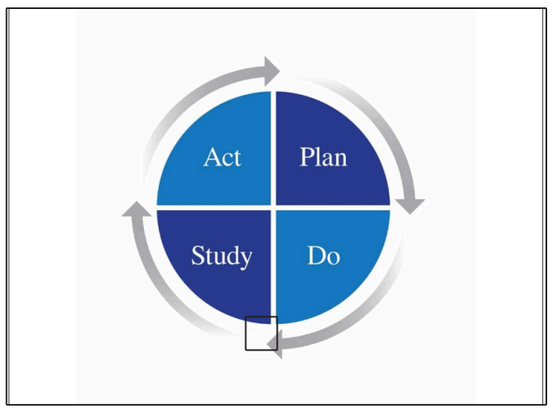
In bug isolation, Fred and Bill used the rule of three. Whenever they had to decide the next course of action, they would identify at least three different alternatives. Comparing and contrasting alternatives helped them to decide what to do next.

Retrospective

1.23 Process improvement

One of the most influential process improvement approaches used in engineering is called the Deming Cycle.

W. Edwards Deming described a system process improvement model using a four-step cycle. This model is known as a PDSA cycle, which stands for Plan, Do, Study, Act.



Using PDSA, practitioners review the results of process change to determine if it is effective. Process changes are identified to encourage improvements and discourage degradation.

- Plan, prepare for change
- Do, execute the plan
- Study, study the results
- Act, adapt based on your findings

1.24 Bug Clusters

Many aspects of software testing can influence process improvement. Testing findings can help identify quality concerns about a product. Test findings can also help identify potential process improvements for all aspects of software development.

When testers identify multiple bugs that have a common cause, then a bug cluster has been identified. By eliminating the cause, multiple bugs can be corrected and future related bugs can be avoided.

To understand the cause of bugs, testers should review with programmers the actual technical work done to correct the defect. Clusters are defined by a common cause, not necessarily by common effect.

1.25 Example of code listening: "agile" retrospective

Retrospective

This is a short experience report by author, Robert Sabourin, about code listening used in an "agile" retrospective meeting.

The team held a retrospective meeting immediately after each sprint. The retrospective meeting was held immediately after the customer's work demo was done in the sprint.

The goal of the retrospective meeting was to review work done by the team in the previous sprint and look for opportunities to self-organize, encourage effective practices, and change ineffective practices. Changes would be implemented in the following sprint. The scrum master facilitated the retrospective meeting and took note of changes to apply to the team's development approach.

The first step of the retrospective meeting related to the work done. The team discussed the following points:

- What worked well?
- ·What did not work well?

For points that worked well, the team took note of practices they'd continue to use.

For points that did not work well, the team discussed alternatives and decided upon changes to their method of operation for subsequent sprints.

The second step of the retrospective meeting related to improvements in programming and testing tasks. The team discussed the following points:

- Any wasted time?
- Any missing tasks?

The team used this knowledge to improve planning in future sprints.

If a task was a waste of time, the team discussed whether it could be split up, avoided, or implemented differently in the future. Missing tasks were added to checklists the team maintained to help in planning future sprints.

The third step of the retrospective meeting related to collaboration. The team reviewed collaboration within the team and with people outside of the team. Excellent collaboration practices were to be encouraged. Weak collaboration practices led to discussions of how to improve such collaboration in the future.

The fourth step of the retrospective meeting related to bugs. The team looked at bugs found and fixed during the sprint. If several bugs had a common or similar cause, the team tried to identify means of avoiding the bugs or catching them earlier; for example, in unit testing.

Some Concluding Remarks

Testers can apply code listening skills to discover many great test ideas.

Code listeners are well positioned to collaborate with their peers in programming, architecture, and implementation, focusing on risks based on what is really changing in the source code.

Code listening can help testers focus their work based on the impact of changes to the source code of a system over and above any knowledge they have about product requirements, acceptance criteria, target environment, and usage models.

The authors' experiences suggest that testers who speak to developers about system structure gain a better understanding of failure modes. For example, what can break when a system is operating?

The authors' experiences suggest that testers who speak to developers about code changes in a system can learn how to identify testing ideas, which can expose unintended consequences.

Technical collaboration between testers and programmers can lead to more efficient testing, troubleshooting, and debugging of a system, which minimizes rework and eliminates unnecessary or redundant testing efforts.

Acknowledgments

Robert Sabourin wishes to thank the thousands of students who have participated in test training courses over the past four decades.

Robert Sabourin and Mario Colina wish to thank many of their professional colleagues who have contributed to their learning when it comes to the value of code listening and the importance of technical collaboration between professionals with different roles and responsibilities in a software engineering project.

References

[1] Myers, et al. The Art of Software Testing. John Wiley & Sons, 2012.

[2] Kaner, Cem, and James Bach. Lessons Learned in Software Testing. Wiley, 2001.

[3] Pólya, George. How to Solve It: A New Aspect of Mathematical Method. Doubleday, 1957.

[4] Sommerville, Ian. Engineering Software Products. Pearson Education, Inc., 2020.

[5] Copeland, Lee. A Practitioner's Guide to Software Test Design. Artech House, 2008.wein

[6] Sabourin, R. Charting the Course Coming Up with Great Test Ideas Just in Time. AmiBug, 2020.

[7] Dijkstra, Edsger, "Programming methodologies, their objectives and their nature." 1969

[8] Freedman, Daniel P, and Gerald M Weinberg. 1990. Handbook of Walkthroughs, Inspections, and Technical Reviews: Evaluating Programs, Projects, and Products. 3rd ed. Little, Brown Computer Systems Series. New York, NY: Dorset House Pub.

[9] Gilb, Tom, Dorothy Graham, and Susannah Finzi. 1993. Software Inspection. Wokingham, England: Addison-Wesley.

[10] Wiegers, Karl Eugene. 2002. Peer Reviews in Software: A Practical Guide. The Addison-Wesley Information Technology Series. Boston, MA: Addison-Wesley.

[11] Fagan Michael, "A History of Software Inspections" in Broy, M, Ernst Denert, and Sd & m AG. 2002. Software Pioneers: Contributions to Software Engineering. Berlin: Springer.

[12] von Neumann, John (1945), First Draft of a Report on the EDVAC.

[13] Kaner, Cem, "SOFTWARE NEGLIGENCE AND TESTING COVERAGE", Software, QA Quarterly, Vol. 2, #2, p. 18, 1995

[14] McConnell, Steve. 2004. Code Complete (version 2nd ed.). 2nd ed. Developer Best Practices. Redmond, Wash.: Microsoft Press.

[15] Vaidhyam Subramanian, Shivashree Vysali. 2021. "Enriching Code Coverage with Test Characteristics." Dissertation, McGill University Libraries. McGill University.

[16] Craig, Rick D, and Stefan P Jaskiel. 2002. Systematic Software Testing. Artech House Computing Library. Boston: Artech House.

[17] Tornhill, Adam, and Michael C Feathers. 2015. Your Code As a Crime Scene: Use Forensic Techniques to Arrest Defects, Bottlenecks, and Bad Design in Your Programs. Edited by Fahmida Y Rashid. The Pragmatic Programmers. Frisco, TX: Pragmatic Bookshelf.

[18] Ambler, Scott W. 2002. Agile Modeling: Effective Practices for Extreme Programming and the Unified Process. Programming, Software Development. New York: J. Wiley.

[19] Kruchten, Philippe. 2004. The Rational Unified Process: An Introduction. 3rd ed. The Addison-Wesley Object Technology Series. Boston: Addison-Wesley.

[20] Schwaber, Ken, and Mike Beedle. 2002. Agile Software Development with Scrum. Series in Agile Software Development. Upper Saddle River, NJ: Prentice Hall.

[21] Schwaber, Ken. 2004. Agile Project Management with Scrum. Redmond, Wash.: Microsoft Press,

[22] Cohn, Mike. 2004. User Stories Applied: For Agile Software Development. Addison-Wesley Signature Series. Boston: Addison-Wesley.

[23] Derby, Esther, and Diana Larsen. 2006. Agile Retrospectives: Making Good Teams Great. The Pragmatic Programmers. Raleigh, NC: Pragmatic Bookshelf.

[24] Deming, W. Edwards. 1986. Out of the Crisis. Cambridge, Mass.: Massachusetts Institute of Technology, Center for Advanced Engineering Study.

[25] Weinberg, Gerald M. 2001. An Introduction to General Systems Thinking Silver anniversary ed. New York: Dorset House.

[26] Seashore, Charles and Seashore, Edith and Weinberg, Gerald M. 2013, The Art of Giving and Receiving Feedback, Smashwords.

[27] JavaTM Platform, Enterprise Edition 6 API Specification. (2011, February 10). https://docs.oracle.com/. Retrieved June 27, 2022, from https://docs.oracle.com/javaee/6/api/index.html?overview-summary.html

[28] Harte, Lawrence. 2008 IPTV Testing. Althos Publishing, Fuquay-Varina, North Carolina

ROBERT SABOURIN
–

Robert Sabourin has more than forty years of management experience, leading teams of software development professionals. A well-respected member of the software engineering community, Robert has managed, trained, mentored, and coached thousands of top professionals in the field. He frequently speaks at conferences and writes on software engineering, SQA, testing, management, and internationalization. The author of "I am a Bug!", the popular software testing children's book, Robert is an adjunct professor of Software Engineering at McGill University. Robert is the principal consultant (and president/janitor) of AmiBug.Com, Inc.

MARIO COLINA
–

Mario Colina has more than 20 years of systems integration and software testing experience in the area of optical networks, wireless, location-based services, MoIP, music delivery, and IPTV-related technologies, just to name a few. He is passonate about software engineering and is an active member of the Context-Driven community. Mario is open-minded and creative in his approach to any project. Has a passion for critical thinking, innovation, and new product ideas, which included spearheading a joint patent initiative for Cloud TV (Patent No.: US 10,362,366). Mario is continually invited as a guest speaker for McGill University's Software Engineering in Practice course.

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

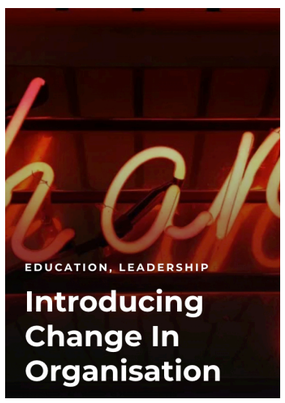
INTERVIEWS, OLD IS GOLD

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.

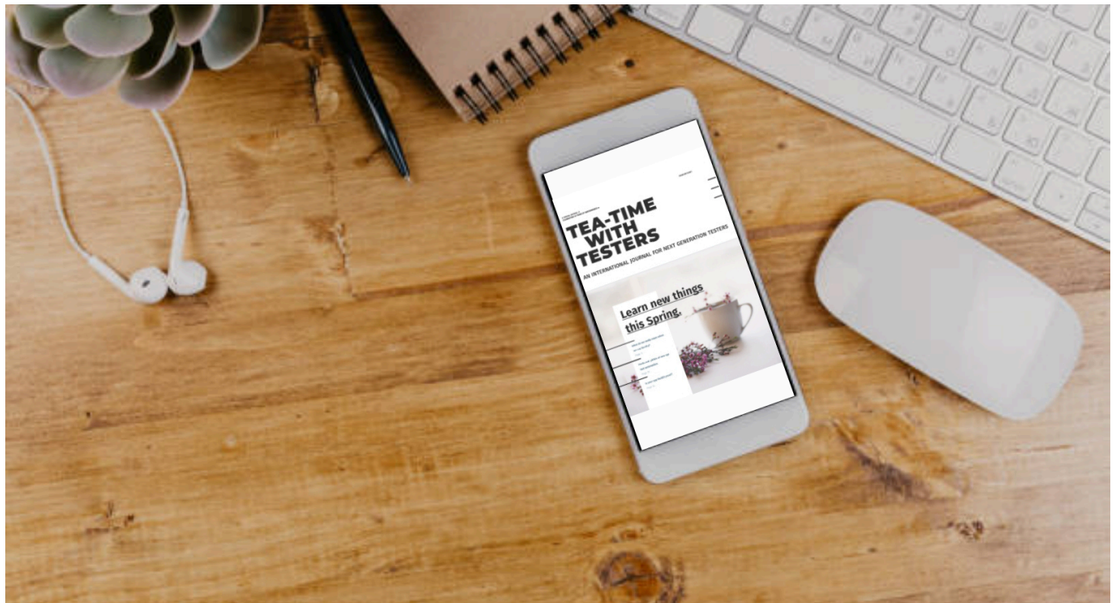
We plan to continue doing that for more articles, interviews and also for the recent issues we have published.

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!



APPROACHES TO CONTRACT TESTING

Recently, I have started working on a new consulting project with a client in the UK. In this role, I am helping them implement contract testing to get better insights into the effects that changes introduced by individual teams on individual services have up- and downstream in a distributed software environment.

Now, most people, when thinking of or talking about contract testing, immediately think of the consumer-driven variant, often referred to as CDCT. However, contract testing is broader than 'just' CDCT. One of the first questions that we typically need to answer, and one that is often forgotten, is

What kind of contract testing approach would be the best fit for our situation?

In this article, I'll present three different approaches to contract testing as well as their respective benefits and drawbacks. I am not going to discuss the merits of contract testing in general in this post. If you're interested in reading more about that, I recommend you have a look at this blog post series instead.

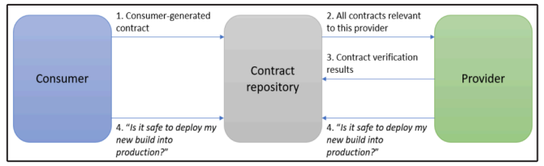
Consumer-driven contract testing (CDCT)

As the name suggests, in CDCT it's the consumer that is calling the shots, so to say. The consumer writes down their expectations about the behaviour of a provider in a contract, then passes that contract to the provider. IT is then the provider's responsibility to demonstrate that they are able to meet all of the expectations expressed by the consumer.

It's important to keep in mind here that a provider often has to demonstrate that they can meet the expectations of many, many different consumers. Each of these consumers hands over a contract with their specific expectations, and the provider has to meet all of them. This does mean that there can be conflicts of interest.

As a simple example, consider the situation where consumer A expects a provider to return a house number in an address as a string, whereas consumer B might expect the same provider to return that same house number as an integer. This is exactly the kind of potential integration issue that consumer-driven contract testing can uncover, and the kind of issue that often slips by unnoticed with more traditional approaches to integration testing. Or when no integration testing is done at all..

Here's what the typical CDCT flow looks like:



In words:

1. The consumer writes down their expectations about provider behaviour in a contract and publishes that to a central repository
2. The provider pulls the relevant contracts from the repository and verifies whether it can fulfill all of the expectations in all of the contracts
3. The provider publishes the verifications results back to the repository
4. Both consumer and provider can query the repository for the latest verification results to see if there are any potential integrations issues, and if it is safe to deploy their next build into production

CDCT is particularly useful in situations where:

- consumers and providers are able to communicate easily to discuss testing situations and work out potential integration issues
- consumers are willing to spend the effort writing the expectations (pacts) and the tests that are required to generate the contracts from these expectations

More situations where CDCT works well, as described by the team behind Pact, one of the leading contract testing tools available today, can be found here.

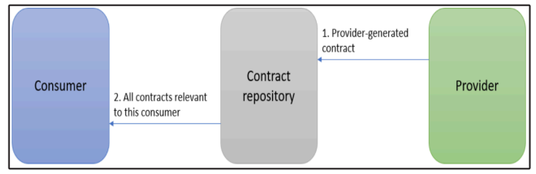
CDCT does not particularly work well in situations where:

- the provider is a public API, i.e., it is hard or even impossible to maintain communication with the provider development team for individual consumer development teams
- the provider is not keen to do contract testing in general, or to listen and adapt to the needs of individual consumers (again, public APIs are a good example here)

More situations where CDCT does not work well are listed here.

Provider-driven contract testing (PDCT)

With PDCT, as you might have guessed from the name, it is the provider, not the consumer, who is 'in charge'. Basically, it comes down to the provider issuing a contract expressing the way they behave and telling their consumers 'this is what I do, deal with it'. A typical PDCT flow is therefore much more straightforward than the CDCT flow we saw earlier:



In words:

1. The provider issues a contract expressing their behaviour
2. Consumers use the contracts issued by the providers to determine whether they can communicate with the provider

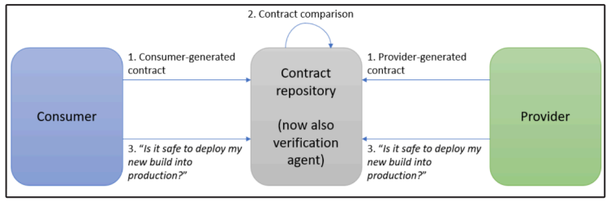
In my opinion, the biggest drawback of PDCT is the lack of a feedback loop, while CDCT's biggest drawback is probably the effort it takes to do it, do it well and keep doing it. This is the reason that recently, a third type of contract testing has emerged.

Bidirectional contract testing (BDCT)

With BDCT, neither the consumer nor the provider is significantly 'in the lead'. Instead, with BDCT, both consumer and provider create their own version of a contract for a specific integration, the consumer contract containing (as in CDCT) their expectations about provider behaviour, the provider contract (as in PDCT) containing a specification of their behaviour.

The main difference between BDCT and the two other flavours of contract testing we discussed earlier is that with BDCT, contract comparison and verification is done by an independent entity instead of either by the consumer or the provider. Both parties upload their contract to this third party, which then compares the two and checks for potential integration issues.

Here's what that flow looks like:



In words:

1. Both consumer and provider upload their contract to the contract repository
2. The repository (which now also acts as a verification agent) compares the contract and checks for potential integration issues
3. Both consumer and provider can query the repository for the latest verification results to see if there are any potential integrations issues, and if it is safe to deploy their next build into production

Currently, the only way I know of to do BDCT is by using Pactflow. If you want to see a working example of BDCT, here's one I created and wrote about recently.

The biggest benefit of practicing BDCT is that the way BDCT is implemented within Pactflow and the wider Pact ecosystem means you can leverage existing technology to generate contracts more quickly, without having to depend on a full-blown Pact implementation.

A drawback for some teams and companies might be that right now, you need to use Pactflow (either the cloud version or an on-premise installation) to be able to practice BDCT.

As you can see, there's more than one way to do contract testing, and each approach has their own benefits and drawbacks. Before you start throwing tools at your integration testing problem, it's therefore a good idea to take a step back and first ask yourself 'what is the best approach to contract testing for our particular context?'.

BAS DIJKSTRA
–

Bas Dijkstra, is an independent test automation consultant and trainer.

He has been active in the test automation field for some 16 years now, and has worked on software testing and automation solutions across a wide range of programming languages, frameworks and technology stacks.

Know more about Bas.

TEA-TIME WITH TESTERS

JOURNAL FOR NEXT GENERATION TESTERS

CONTENT PREVIEW : ISSUE 02/2022

MORE AWESOMENESS IS ON YOUR WAY THIS SEASON!

01

TESTING THE RECOMMENDATION SYSTEMS

The age of digital transformation has brought with it a wealth of information. However, filtering it to be usable is can be highly challenging. It is now possible to understand patterns in user behavior and then correlate it with the other user's behavior to predict and help in the decision- making process. Do not miss this exclusive article by Soumya Mukherjee in next issue.

02

TEA AND TESTING WITH JERRY WEINBERG

We are bringing back the treasure of knowledge that Jerry Weinberg has left behind for us. More awesomeness on its way....

03

TESTING AND PREVENTION- THE ILLUSION

Thought provoking article by Paul Seaman that will make you question some popular beliefs and disbeliefs. Fasten your seatbelts.

INTERNATIONAL JOURNAL FOR NEXT GENERATION TESTERS

TEA-TIME WITH TESTERS

THE SOFTWARE TESTING AND QUALITY MAGAZINE

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