THE INTERNATIONAL MONTHLY FOR NEXT GENERATION TESTERS

Tea-time with Testers

January 2018

Articles by –

Jerry Weinberg

T Ashok

Marcus Noll

Helena Jeret-Mäe

Nadia Cavalleri

Jaan Jap Cannegieter

and others...

TEA-TIME WITH TESTERS

Created and Published by:

Tea-time with Testers.
B2-101, Atlanta, Wakad Road
Pune-411057
Maharashtra, India.

Editorial and Advertising Enquiries:

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

Lalit: (+49) 15227220745
Pratik: (+49) 15215673149

Editorial

Here comes 2018!

Another great year has passed and we have stepped into 2018 already. Awesome!

2017 has been an interesting year for me as a tester and as a person. At one hand I got to learn a lot of interesting things, I participated in multiple projects and gathered more experience, became father while on other hand I had to put great efforts managing multiple things together, personally and professionally.

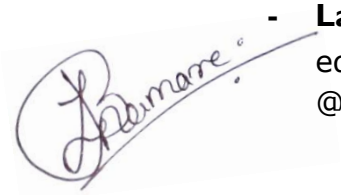
But as they say, every situation teaches you something and offers pleasure of its own, I indeed enjoyed all the moments 2017 had to offer me. And I am grateful to life, people who supported me and especially you dear readers.

While this year we have not been as regular as previous years, I appreciate that most of you still stayed with us and showed great interest in coming issues. Thank you.

Good news is, we are back again with some truly amazing articles. And we will do our best to offer you the best editions in 2018 too.

On that note, I thank you all for staying with us this far and wish you a great new year 2018!

Sincerely Yours,



- **Lalitkumar Bhamare**
editor@teatimewithtesters.com
@Lalitbhamare / @TtimewidTesters

QuickLook

The Year That Software Bugs Ate The World

By Harry McCracken via www.fastcompany.com:

In 2017, bugs banned people from Twitter, secretly recorded them in their homes, and even caused a train crash. Is there anything they *can't* do?

In 2017, it was fashionable to stress over the prospect of machines getting so smart that they render humans obsolete or maybe even decide to kill us all.

Look on the bright side, though: This also turned out to be a year that provided an inordinate number of reminders that what computers do is follow instructions given to them by people. And people have a tendency to write buggy software. When it fails, it can be startling, alarming, irritating, or darkly funny—or, sometimes, all of the above.

Herewith, some, um, highlights from the year in bugs, all of which involve defects that were fixed, sooner or later.

Read complete report directly at the source.

What Is Quality? Why Is It Important? – part 2

Helpful Hints and Suggestions

Of course you can't do a perfect job of identifying all potential users of your software and determining what they value, but that doesn't mean you won't benefit from trying. In fact, you'll probably find it beneficial just to try doing it in your head. Once you've experienced those benefits, you may decide to interview at least a few major users to find out where their values lie.

Because of the conservative nature of culture, attempts to change are always met with "resistance." You will be better able to cope with such "resistance" if you recognize it as attempts to preserve what is good about the old way of doing things. Even better will be to begin a change project by acknowledging the value of the old way, and determining which characteristics you wish to preserve, even though changing the cultural pattern.

Summary

1. Quality is relative. What is quality to one person may even be lack of quality to another.
2. Finding the relativity involves detecting the implicit person or persons in the statement about quality, by asking, "Who is the person behind that statement about quality."
3. Quality is neither more nor less than value to some person or persons. This view allows us to reconcile such statements as, "Zero defects is high quality." , "Lots of features is high quality." , "Elegant coding is high quality." , "High performance is high quality." , "Low development cost is high quality." , "Rapid development is high quality." , "User-friendliness is high quality." All of the statements can be true at the same time.
4. Quality is almost always a political/emotional issue, though we like to pretend it can be settled rationally.
5. Quality is not identical with freedom from errors. A software product that does not even conform to its formal requirements could be considered of high quality by some of its users.
6. Improving quality is so difficult because organizations tend to lock on to a specific pattern of doing things. They adapt to the present level of quality, they don't know what is needed to change to new level, and they don't really try to find out.
7. The patterns adopted by software organizations tend to fall into a few clusters, or subcultures, each of which produces characteristic results.
8. Cultures are inherently conservative. This conservatism is manifested primarily in

    a. the satisfaction with a particular level of quality
    b. the fear of losing that level in an attempt to do even better
    c. the lack of understanding of other cultures
    d. the invisibility of their own culture

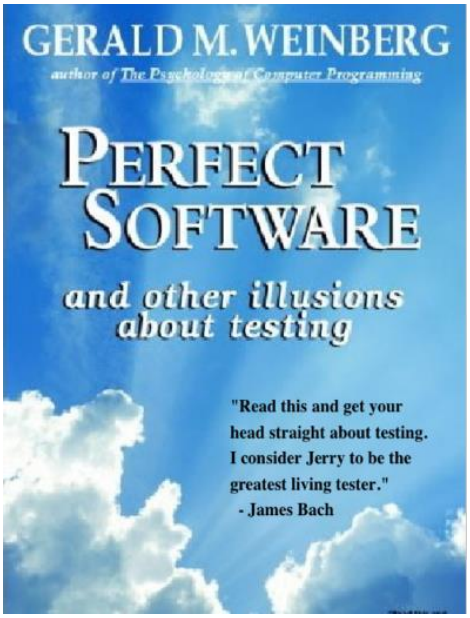
Practice

1. I wrote to Doug Brent, telling him how grateful I was and showing him the two erroneous cases, but haven't gotten a reply so far. I wouldn't mind if Precision Cribbage were corrected, but I wouldn't pay very much for the corrections, because their value was reduced once I had an approximate cribbage program with which to play. Discuss how the value, and thus the definition of quality, changes for a particular software product over time, as early versions of the product, or competing products, come into use.

2. Produce a list of characteristics that an organization might lock onto when standardizing on a given hardware architecture.

3. What evidence can you produce to indicate that people in your organization are indeed satisfied with the level of quality they produce? How does the organization deal with people who express dissatisfaction with that level?

If you want to learn more about testing and quality, take a look at some of Jerry's books, such as:



Curious to know why you should get 'People Skills' bundle by Jerry? Then check this out

Biography

Gerald Marvin (Jerry) Weinberg is an American computer scientist, author and teacher of the psychology and anthropology of computer software development.

For more than 50 years, he has 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.**

To know more about Gerald and his work, please visit his Official Website here .

Gerald can be reached at hardpretzel@earthlink.net or on twitter @JerryWeinberg

Jerry Weinberg has been observing software development for more than 50 years.

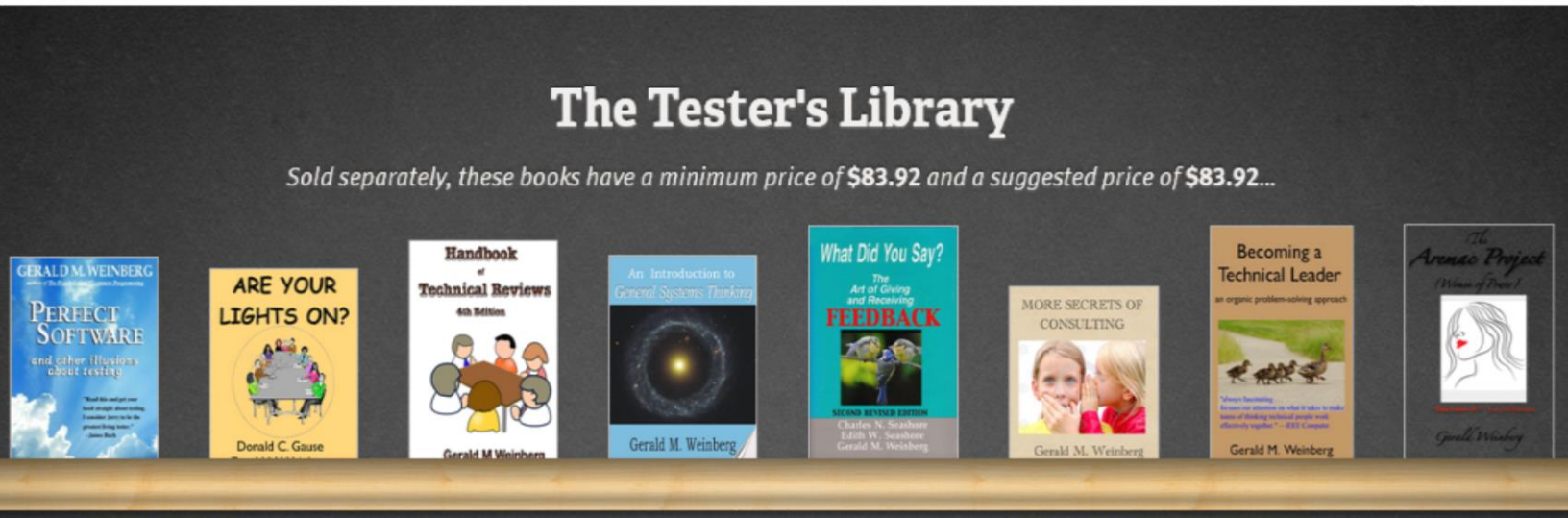
Lately, he's been observing the Agile movement, and he's offering an evolving set of impressions of where it came from, where it is now, and where it's going. If you are doing things Agile way or are curious about it, this book is for you.

Know more about Jerry's writing on software on his website.

TTWT Rating: ★★★★★

The Bundle of Bliss

Buy Jerry Weinberg's all testing related books in one bundle and at unbelievable price!



The Tester's Library consists of eight five-star books that every software tester should read and re-read. As bound books, this collection would cost over $200. Even as e-books, their price would exceed $80, but in this bundle, their cost is only $49.99.

The 8 books are as follows:

- Perfect Software
- Are Your Lights On?
- Handbook of Technical Reviews (4th ed.)
- An Introduction to General Systems Thinking
- What Did You Say? The Art of Giving and Receiving Feedback
- More Secrets of Consulting
-Becoming a Technical Leader
- The Aremac Project

Know more about this bundle


Speaking Tester's Mind
- straight from the author's desk

Heuristics for Mushroom Picking (and testing)

- by Helena Jeret-Mäe

Mushroom picking has been part of my autumns since I can remember. At least here in Estonia our hunter-gatherer roots still run strong. Wandering around the forest with a bucket and knife in hand while enjoying the tranquility and calm of a pine forest is one of the most enjoyable moments before the doom and gloom of winter.

I find mushroom picking to be a meditative activity where one part of my mind focuses on mushrooms while the other one meditates on whatever comes to mind. So this time I found myself pondering the similarities between mushroom picking and testing.

And yes, I found them. Why I wanted to write about it is because heuristics and oracles aren't that easy to grasp for beginners or people interested in testing. I've taught several workshops and have coached people about these concepts. So at least for people who are familiar with mushroom picking or other foraging activities, comparing mushroom picking to bug hunting and testing could be very relatable.

So here goes…

Choice of location vs. product coverage

Typically, I go to a handful of forests that I know like the back of my hand and I know what type of mushrooms I can find there. Since my mission is to effectively collect a sufficient amount of mushrooms, going to the same locations again and again satisfies my needs quite well. However, if I wanted to collect new types of mushrooms, I'd have to take more risks: scout for a different forest, ask people around (though the secret locations aren't divulged that easily if at all), do more research but I may not end up with any mushrooms after all.

In testing, visiting the same parts of the product over and over again would mean that your test coverage of the product is limited in terms of scope. Unless the product is very unstable, you're also not

likely to find anything very new and exciting there. But if you spend a lot of time in a limited area, you also get to know it really well, so your testing could go in more depth.

It's always a trade-off when choosing how to spend your time, but we know what to focus on based on our mission: do we need to discover new information (mushrooms) and test previously uncovered parts of the product or not?

Making preparations

When planning to go mushroom picking, I typically keep an eye on rainfall and temperatures, and also the media since news about "good" or "not so good" mushroom season get published in major online news sites. Going mushroom picking later in the autumn when more and more leaves turn golden yellow may also cause a lot of "false positive" alarms (you think you saw a mushroom but… it was a leaf). In testing, I also try to be in the know about recent changes in the software and any rumors and opinions floating around. It's data that I can use to decide if and what needs testing before I set out.

I make sure I have my rubber boots, proper knife and a bucket or a basket. The basket shouldn't be too small (too many interruptions when going back to the car to get another one which wastes time) or too large (too heavy and difficult to carry it around as it fills up). Some people also carry a little brush to clean the mushrooms on the spot (I can never spare that time while in the forest, so I rather do it afterwards). Similarly, I will think of tools I may need when approaching a testing problem. Do I need any or do I need help with some of those tools? Do I know the best options available or should I do some research?

Know your oracles

One of the first things I can remember that I was ever taught about mushrooms was how to recognize the poisonous ones. Picking mushrooms really is a matter of life and death: if you pick the wrong ones and actually eat them, you will die. Period. Hence knowing your oracles for recognizing poisonous mushrooms is really important.

The first thing I was taught as a child was that a poisonous mushroom has a ring around its stalk. This is true for fly amanita which also has a very easily recognizable cap) and other types of amanitas. Hence, it's ***generally*** a good heuristic to use to avoid such mushrooms.

However, one of my favorite mushrooms which is not poisonous and can simply be pickled or frozen (after heating them on the pan to get rid of the water) has a ring around its stalk. It's called gypsy mushroom (Cortinarius caperatus). Many a times I've seen other types of mushrooms being picked but gypsy mushrooms are left behind. Well, I don't mind, really…

The "ring means poison" heuristic is typically helpful for survival but if it's the only oracle used, it doesn't really help us fully evaluate the situation. There are more characteristics for identifying bugs (and poisonous mushrooms) than just one (cap color, type, stalk type, etc.). Therefore, we need to ask ourselves if we have more than one piece of information that could be useful when recognizing a bug, and what could be missing that would change the evaluation.

On the one hand, sticking to this specific heuristic means minimizing risks. On the other hand, sticking to very few oracles, I am able to recognize a limited amount of edible mushrooms. I know that I, too, walk past numerous perfectly edible mushrooms but since I haven't taken the time to learn more and taken the risk of venturing into new territories, I leave these mushrooms behind.

We need to take more risks in testing, question and learn and push ourselves. Because, it may be a matter of "life and death" in a different way.

Quantity and quality

So if you walk out of the forest with the biggest bucket of mushrooms, are you the most successful mushroom whisperer? Well, it depends. It depends on the quality of mushrooms you've picked. If it's not the best season and worms have wreaked havoc, you may not find many mushrooms in acceptable condition, so you may start to lower your notion of quality. Maybe you're not inspecting the mushrooms very carefully and trying not to see the wormholes… Maybe you're trying to tell yourself this one would still be good… But when you get home, you may end up discarding a large percentage of mushrooms when you finally sit down and evaluate them. Also, if the quantity is large, it is takes many hours to clean, chop and prepare. Different mushrooms need different treatment (some need to be boiled, some need to be soaked before boiling, etc.)

Similarly, it doesn't mean that you're a great bug whisperer when you find a lot of bugs. Some may not be bugs at all. Some may be low risk and unimportant. Some may not be properly described. And it may happen that if you bring too many issues on the table and ask people to inspect them one by one, you bring the team's progress to a halt.

When mushroom picking, you will need to inspect the condition of the cap (mushrooms age differently but you can tell if a mushroom is old), the stalk for wormholes (sometimes only the lower portion is bad but you need to cut it shorter to see if the rest is fine; sometimes you may also need to cut the cap). And sometimes you need to leave that mushroom behind.

While testing you also need to continuously evaluate what you find, dig deeper, not take things at face value, and sift through the information to take back what is really valuable.

What to do when you've found one

The heuristic to use when you've found a mushroom is "look for others of the same kind" nearby. Chanterelles are a good example of this. You also need to know they typically hide themselves in the moss, so you need to peel your eyes and dig around a little. But sometimes you can't find the others because you're standing on top of them. Or you placed your bucket on them when bending over to pick the one that caught your eye. Then I use the "glance over your shoulder" heuristic when I walk away from a spot. Sometimes sunlight changes what is visible and what is not, so you need to check the place under slightly different angles.

If you're a beginner mushroom whisperer, take note of the location where you found the mushroom. Observe the surroundings. What is the undergrowth like? Is it taller, does it contain moss or also grass? Is it on the higher ground or lower? Is it close to certain types of trees? This is how you develop a nose for mushrooming, picking up on characteristics and their combinations, which is what will help you navigate even an unknown forest in the future.

In testing we pay attention to similar things. Where are the bugs most likely to occur? What characterizes the location in the product? What kind of issues cluster together? Over time you develop a nose for bugs, too. But, only when you pay attention and develop your observation skills.

And finally, I find that it is difficult for me to explain everything about how I find mushrooms (and how I find bugs). There's tacit knowledge that I've accumulated but that I can start to make sense of when I observe myself. Also, it's sometimes hard to explain why two people go into the same forest and come out with a different amount of mushrooms. Is it luck? Or is it the skills developed throughout the years?

To her own great surprise, **Helena Jeret-Mäe** became passionate about software testing after stumbling into it via technical writing.

She has worked on testing medical practice management software, and has built and lead a testing team. She has also worked as a competence head and she's currently a consultant focusing on process quality and team collaboration. Helena loves to be part of the testing community because of the countless learning opportunities, so she can be found discussing testing in a pub with some testers, at testing conferences, or reading about testing in the corner of a café.

Helena tweets as @HelenaJ_M and blogs at thepainandgainofedwardbear.wordpress.com

Having Fun with your Testing Team

- by Nadia Cavalleri

If you feel that your testing team has a low level of interaction and motivation, it is time to introduce some activities that could help you to exchange knowledge, share experiences and strengthen the links between team members.

In this article I will tell you about different activities that I introduced in my testing teams and how it went. Dedicating between 2 and 4 hours per month for this type of activities, you could get excellent results. We did it and we reach them. These techniques could be performed alone or in combination. It is important not to do them randomly. Previously, you have to analyze the needs of the team at that time.

Here are the activities we did:

Ice breakers: which are small and quick games with different goals such us learning each other's names, finding out interesting things in common, helping people to begin new relationships with colleagues, etc. After performing an ice breaker people are more relax and have more predisposition to work together on a new activity. It is a good way to start a meeting, especially when there are many new people in the team or when the level of interaction between them is very low. One of the icebreakers that we perform was "Finding 10 things in common".

Crowd testing: we did it on an ongoing Project. This activity consists on giving a short introduction of one of our current Projects to the rest of the team. Then testers worked in small teams to test it. We try that each team has people with different seniorities and skills. It was an enriching experience because testers juniors learned from seniors and we also found bugs that the tester of the project never saw. This is not because the tester of this project has a bad performance. Sometimes we have blind spots because we are very accustomed to the application under test.

- Brainstorming session: which is a tool that greatly helps in creative processes. At the beginning, you will choose a topic to be debated, then you have to promote the generation of large amounts of ideas. All members of the team can make their contributions without censorship. The most important thing here is the quantity, not the quality of the ideas. When we have many, we can move to the next step. Now you can evaluate the quality of those ideas and decide which of them will be carried out. We use this technique for different purposes, for example, when someone has a problem and we want to help him or her to think solutions. We also use it to enrich the process of decision making by providing different points of view and making a better analysis of the alternatives.

- Role-playing sessions: we try the testers happen through everyday situations but in a controlled environment to practice how to respond. For example, when they have to negotiate testing estimations or automation needs. It will let them to build confidence, gain experience and get feedback immediately. If you want, you can also record the sessions. It is a very good material for the person who is playing the role.

- Coding dojo: we introduced it to learn test automation tools. A Coding Dojo is a programming session based on a simple coding challenge. Originally, it was intended for development teams but we adapt it to test automation teams because in some way it implies developing scripts. The goal of a coding dojo is learning, teaching and improving something that could be a technique, a programming language, a framework, etc.

- Pecha kucha, lightning talks or courses: we use some meetings to share the knowledge that we learned. These trainings can be in different formats. The traditional one is a person who dictates a course but you can also introduce a session of short talks where each speaker briefly presents a topic that he or she has learned in the last time. Pecha-kucha and lightning talks are examples of this kind of format. Topics may be varied. It is not necessary to talk about testing. There are many different things to learn that you can apply to your daily work for example How to manage your time, Motivation or the 5S technique.

- Peer review sessions: we exchange people between projects to evaluate them and propose improvements. This will allow you to deploy good practices and learning from one project to another. If the company has a defined testing process, this activity will also allow you to detect deviations, to propose improvements to the process and to train people who do not know the defined process.

- Case analysis: we carry out this activity to analyze cases of success or failure and share the lessons learned. This activity can be done when you win or lose a business proposal or when a project is canceled or when it successfully ends.

I hope you are encouraged to introduce these and other activities into your teams. Do not hesitate to share your doubts or experiences. Let's transform our testing teams in funny, interactive and collaborative ones!

Nadia Soledad Cavalleri lives in Buenos Aires, Argentina. She is an information systems engineer (2008) and psychologist (2012). In 2009, she obtained the IBM – Rational Functional Tester certification.

Nadia has been working in testing for more than twelve years. Nadia was the south American Judge at the Software Testing World Cup in 2014 and 2016. She also was speaker in TestingUY, Agile Testing Days, QS-Tag and QA&Test. And she is part of the evaluation comittee in ExpoQA. She has also delivered courses at schools, universities, and companies.

Contact Nadia on nscavalleri@gmail.com or on Twitter @nadiacavalleri on LinkedIn /in/ncavalleri

The Agile Testing Pyramid: Not so much about Tools But more about People And Culture

- by Jaan Jap Cannegieter
- Lucas Sikking and
- Martijn van Werven

Some concepts are brilliant in their simplicity but very complex when you apply them. The Agile Testing Pyramid from Mike Cohn is such a concept, see box 1. The authors of this article were involved in implementing the agile testing pyramid in a big governmental organization in the Netherlands. In this article we look at the influence of implementing the pyramid on test goals, tooling, type of testers and culture. We found out that implementing the agile testing pyramid is not only more difficult than it might look, we also found out that implementing the agile testing pyramid is mostly about people and culture.

The organization where we implemented the agile testing pyramid is a big governmental organization in the Netherlands. Most of the business processes are executed without any human intervention. It doesn't need any explanation that testing is a very important in this organization. This organization has a long history of testing and test automation. About four years ago this organization decided to implement Scrum in a part of the organization. The testers became part of the multidisciplinary teams and beside functional correctness and reliability speed became also an important aspect of testing. So this organization started a year after the introduction of agile to optimize test automation using a

functional test tool. In the next one-and-a-half year the regression test set was build, reliability, performance testing and test data management were the next improvement points. All tests were end-to-end tests; there was too little focus on testing lower technical layers directly.

This organization had different challenges and goals with the introduction of the Agile Testing Pyramid. With the pyramid support and trust between development team and the product owner/business should be created. Especially for the more technical testing layers, it can be a challenge to advocate its value right from the start. Secondly they wanted to display the result and value of these technical tests in an easy and for everybody understandable way. This way everybody could keep track of what is tested in which layer of the pyramid. The third challenge was how to make sure that the knowledge of the different testing layers, and how to test them, is broadly shared so testing is not dependent on any one person. As these challenges cannot be tackled by any single person or for that matter, any single discipline, good communication and sharing of knowledge is key.

All new projects had to implement the testing pyramid. For running projects considerations whether to implement the pyramid or not where how much time the organisation wanted to invest in the particular project, if they could reuse existing test cases and if they had the right knowledge in the project to implement the pyramid. The testers on the project were most definitely competent; they just lacked experience in test automation. So some of the running projects implemented the pyramid, some didn't.

The original agile testing pyramid knows three levels: unit tests, services and UI tests. In this organization we expanded the agile testing pyramid to four levels: unit test, integration test, non-functional test and functional test. The lowest levels are the unit and component level. A unit is the smallest testable piece of software; a component is a unit plus the interfaces to other units. The third level of the pyramid in this organization was the integration layer. Here the API-testing was done. The fourth level of the pyramid in this organization is non-functional testing, in this organization mostly security testing and performance testing. The fifth and last level is functional UI testing. The agile testing pyramid in this organization was mainly focussed on automated testing, beside the automated tests there was manual testing, this was mostly done by means of exploratory testing. This could be seen as part of the fifth level or as a separate, sixth level. This kind of testing will not be discussed in this article. Our experiences with the application of the agile testing pyramid are divided in test goals, tooling, types of testers you need and culture.

Test goals

The test goals on the first levels, unit test, are white box tests and focused on the correct technical operation of the software units including the interfaces between the units. Technical field validations and integration with the databases is also tested here. These tests are highly technical. Of course business logic is also important on this level, but only small parts of this logic is tested on this level. Complete processes won't be tested yet.

On the integration level, the second level of the pyramid in this organization, the API's between different systems are tested. This organization uses an Enterprise Service Bus (ESB) to integrate the different components (systems, frontend, backend, and databases). On this level SOAP files (Simple Object Access Protocol) are used to exchange structured information between the different components. On this level we tested if the different components processed the SOAP files correct.

On the third level, the non-functional level, we recognized different test goals. The most important aspects in this organization are security and performance. Such kind of testing was done by the development teams to the maximum possible extent, but often done after the sprint. The Scrum theory says all testing should be done by the team to be able to deliver a potentially shippable product, but given the level of specialization of these tests this was not always possible. Usability for the parts of the system where clients (citizens of the Netherlands) interacted with the system was designed and tested by a separate team and was not part of the pyramid. Usability for the employees who were using the applications was not tested separately.

The fourth and last level of the pyramid was focussed on testing the business logic, end-to-end testing. This could be seen as an end-to-end test. Here complete processes and functions were tested. On this level no distinction is made between different layers of the system or between different systems. The test goals on this level are the business processes, business logic and the question if the system was in line with the requirements.

Tooling

We needed different kinds of tooling on different levels of the pyramid. For unit testing and component testing unit test frameworks like JUnit, Test Explorer (part of Visual Studio), Karma, Protractor and Jasmine are used. The technical orientation of these test require a framework that supports this. Test Driven Development can very well be applied on this level with these tools.

The tools used on this level are usually tools like SOAP UI, but other frameworks like Tosca, Protractor and Jasmine can be used here as well. In this organization mostly Tosca, Protractor and Jasmine were used because these tools were also used on other levels.

For non-functional testing, the fourth level of the pyramid in this organization, very specific tools were used. Examples of tools we used are NeoLoad for performance testing as well as Fortify and GuantIT for security testing.

On the last level of the pyramid UI-oriented tools like Tosca, Cucumber, HP UFT, Ranorex and Selenium are normally used. This organization choose to use Tosca and Cucumber. The application of Cucumber made it possible to apply Behaviour Driven Development (BDD).

We came to the conclusion that we needed different tools on different levels of the pyramid. For instance: when you find problems with performance and reliability with functional testing you probably need other tools to study and solve this problem. We didn’t find a toolset that supports the whole pyramid. So organizations that apply the pyramid will need knowledge and experience with different tools, which could be an issue in small organizations or small projects. On our organizations we have different knowledge owners for different tools.

Types of testers

In the eyes of non-testers testing is a specialism. Given the different levels of testing in the pyramid we came to the conclusion there are at least three types of testers. The first type of testers are highly technical testers with good knowledge of programming. These testers are (former) programmers or testers with the technical knowledge to act like a programmer. These testers tested on the first three levels: unit test, component test and integration test. It seems reasonable to let one or more of the programmers take this role in the team. But ideally these technical testers also have knowledge and experience with test techniques and coverage techniques. Knowledge of the specific tooling is relatively easy to obtain. In most of the teams in this organization testers who learned how to program in the used language (Java, C#) took this role, not the programmers.

The second type of testers are the functional testers. They are more business oriented and think in functionality (what) instead of technique (how). Although business processes and test design techniques are there core competences, they also have basic knowledge of programming, databases et cetera.

Ideally, testers should master both the technical and the functional field. These testers are truly multidisciplinary: they can do programming tasks, technical testing tasks and functional testing tasks. But given the difference in knowledge and orientation these testers are rare. This led to the two different kinds of testers in the team: technical testers and functional testers.

The last type of testers are specialised testers, like security testers, performance testers et cetera. These testers test the non-functional and are specialists in the particular area and not so much testers. Test design techniques and coverage techniques are not that important for them, although the knowledge is beneficial for them as well.

Nowadays, many organizations are looking for testers with programming skills, this organization included. In our experience we need both testers with programming skills as well as testers with functional skills. These different kind of testers have different skills and look differently to problems. So ideally you have both in your project.

Culture

When you look at the agile testing pyramid from a distance you could draw the conclusion technical knowledge and test automation is the key aspect of the pyramid. There is no denying that this is the case, but we found out that the organization culture is more important in implementing the pyramid. To implement the pyramid, much of the testing effort is focussed early in the development process. This means less time to develop, and testers generally don't have enough technical knowledge. When you start using TDD and BDD this gives you the opportunity to come up with non-technical test cases that can be discussed with the business, but to execute them you need deep technical knowledge. So in the end most testers need technical skills. In this organization there were more functional testers compared to technical testers. The temptation to let everybody program who can program and only test it on a functional level (apart from some superficial unit testing) is big. But this inevitably least to the

inverted pyramid, a situation you want to avoid. But that means realizing less functionality so that programming capacity is freed for testing on the lower levels of the pyramid. This requires a different kind of mind-set in testers. It means that unit tests, component tests and integration tests should be prepared and executed before the software is released for functional and non-functional testing. And really, really stick to that. It even means disapprove the result of a sprint because the lower level tests are not done. And this comes back to culture.

Implementing the agile testing pyramid is much more than a technical implementation or a mind-set. The test process, toolset, testers' skills and (both organizational and project) culture need to be in line to be successful. Peter Drucker said 'Culture eats strategy for breakfast'. This also turns out to be true when it comes to implementing the agile testing pyramid.

The agile testing pyramid

The Agile Testing Pyramid is an agile test automation concept developed by Mike Cohn. In the original concept the automated tests are divided in three levels: Unit test, service tests (later often called API tests) and User Interface tests. A very important insight Cohn gives us is that automated unit tests are the cheapest tests and automated UI tests are the most expensive. Service tests are in between. The agile testing pyramid it later extended by others with manual exploratory tests, which are more expensive compared to automated UI tests. So from a costs point of view the basis of testing in an agile context should be much unit tests, above that a little less service tests and the on top level of the pyramid some automated UI test. On top of the pyramid you find the exploratory tests. See figure one.

Different participants see that this is not the current practice; often there are only a few unit tests and most of the test automation effort is spend on automated UI tests. On top of that we see the manual exploratory tests. This is sometimes called the Agile Testing Ice Cream, figure two.

For more information: Succeeding with Agile: Software Development Using Scrum. Mike Cohn, Pearson Education, 2010

The Evolution of the Testing Pyramid, James Willet, http://james-willett.com/2016/09/the-evolution-of-the-testing-pyramid/

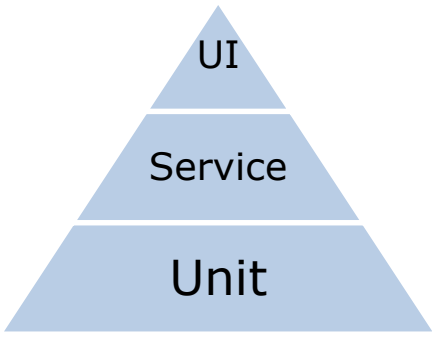


Figure 1: Agile Testing Pyramid

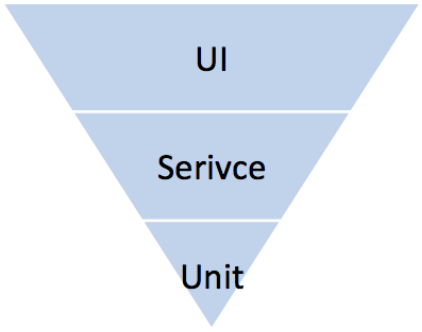


Figure 2: Agile Testing ice cream

Jan Jaap Cannegieter

Lucas Sikking

Martijn van Werven

Jan Jaap Cannegieter – Jan Jaap is Principal Consultant at Squerist, he had 20 years of experience in testing, requirements, quality assurance and process improvement. He published several books and articles on these subject and speaks regular on (international) conferences.

Lucas Sikking – Lucas is a Test Automation Engineer at Squerist with 7 years of experience in Testing and test automation. He has a background in linguistics, and in his work he focusses on how technical details can be translated in a easy to understand way. In his free time he can be found travelling or trying to speak a language that's still on his to-learn list.

Martijn van Werven –Martijn is a Test Automation Engineer at Squerist with over 10 years of experience in Testing and test automation. When not automating everything he possibly can, he can be usually found driving or fixing his motorcycles.


In the
school of
Testing
for your better learning & sharing experience

VIP BOA – A Heuristic for Testing Responsive Web Apps

- by Marcus Noll

It can be quite challenging to test web applications that are built by using the responsive web design [1] (RWD) approach. This heuristic is designed to help anyone involved in that. Some test heuristics come with a mnemonic - which in this case is VIP BOA, representing the four main categories **VI**ew**P**ort, **B**rowser, **O**S, **A**pplication. Here is the cheat sheet, followed by a detailed description for each of those categories. Happy testing!

Viewport	Browser	OS, hardware, device	Application
Start with the smallest	Leave the happy path early	Change device orientation	Localization
Check at the boundaries	Observe the browser console	Resize window	Examine z-indexed elements
Think ALL screens	Zoom in/out	Check at various DPR	Check HTML <head>
	Don't rely on the private tab	With and without scroll bars	Observe the network traffic
		Change input device	Look closely at the font
		Check on lower end	
		Throttle/cut connection	

Viewport size

Start with the smallest

Take a look at the minimum viewport width your application supports. And then start testing there. You will be amazed how many issues that uncovers: Headlines not fitting, elements overlapping, etc. People are often surprised by these issues: They are using current devices for testing which tend to have quite big displays and therefore totally forget to cater for the smallest supported display. To give an example: Let's say the minimum viewport width supported by your application is 320px. Exclusively designing and testing with current mid-range Android and iOS devices in mind (360px - 375px) would most probably lead you into the trap described above. You should definitely make your team aware of this topic to prevent design and/or implementation flaws in the first place.

For a quick check, you could always use the developer tools of your browser. This way, you are able to set the viewport to the exact desired size. It is only a matter of a few clicks in most developer tools (e.g. Chrome, Firefox and Safari). [2]

Check at the boundaries

You should always have boundary analysis in your tool-belt when testing something. And you can also apply it in this context: Look at the viewport ranges/breakpoints that are defined by the media queries in your application. Then set your viewport to the min/max values of these ranges and start testing respectively.

To give you a practical example, let's pretend your application's media queries define the following viewport ranges:

- 320px - 480px
- 481px - 767px
- 768px - 991px
- 992px - 1199px
- >= 1200px

This would mean you would have to set the viewport width to 320px and test. Then set it to 480px and test. Set it to 481px and test – set it to 767 and test – set it to 768 and test – and so on.

The Chrome Developer Tools provide an excellent interface [3] that speeds you up in that process. It displays the viewport ranges based on the media queries defined in your application and allows for easily setting the viewport width to their minimum and maximum values.

Think ALL screens

Responsive web design is not only about **small** screens, but also about **all** screens.

Remember, a desktop user is always able to resize their browser window to just 400px width, even on a 1920px wide display. Also, think of "new" device classes: There are desktop devices out there that come equipped with a touch screen, breaking up with the classic dichotomy between "desktop/click" on the one hand and "mobile/touch" on the other. When creating concepts and designs, always keep in mind that your design canvas is being reduced vertically by browser chrome (i.e. address bars, buttons) in the real world. Not to forget the software keyboard that appears when in an input form. Also, take care of that headline copy that is looking gorgeous on your 1920px wide display: Does it fit on a 320px screen as well? And is this also true for the French translation, which might consist of more characters?

Do not focus on either small screens or large screens and do not make false assumptions. Responsive web design is here to cater for all.

Browser

Leave the happy path early

Carefully choose the browsers in which you are doing your first rounds of tests.

The first one should meet each of the following three criteria: It has a large share among users of your application **AND** it is the preferred browser of the developers in your team **AND** it is considered as being "modern". Opinions on what makes up a modern browser differ, so you may want to check an objective and timeless proposition [4]. By doing your first round of tests only in this one browser you have a high chance to uncover actual issues/gaps in the implementation (and not just some cross-browser problems).

After that, leave the happy path immediately. Pick the browser for your second round of test, which only has to meet two criteria: It has a large share among users of your application **AND** it is notorious for quirkiness. Then start testing there. With the findings from your tests in the previous browser you can now easily identify cross-browser issues. It is valuable to know these issues at such an early stage. On the one hand, because this gives your team the chance to assess whether fixing these issues requires bigger changes in your application. On the other hand, because you only have to retest in a few browsers after the early issues have been fixed. Imagine the mess your team would face if the bad browser were tested last. You would have to fix all the issues and retest the fixes in all the other browsers.

The next browser pick should meet again two criteria: It has a large share among users of your application **AND** it runs on a device-class different than the previous browsers. For example, if the previous two test rounds have been executed on desktop devices, a small screen touch device will be a good addition.

After that, iterate through all the other browsers.

This risk-based approach is inspired by the article "High-Impact, Minimal-Effort Cross-Browser Testing" on Smashing Magazine [5].

Observe the browser console

Errors in the browser console might not have a visible effect while testing an application, so they go unnoticed way too often. Maybe an error is being thrown for a certain part of your application, but you do not notice that because you are actively testing something else or the error just does not have a visible impact. But errors are not the only things you should care for: Maybe a warning is being thrown because a component or API call has been deprecated and you will need to adjust something in your application. Therefore, it is highly recommended to always keep an eye on the browser console (pick the most verbose log level). And that goes for each and every browser your application supports.

Zoom in/out

Zooming in and out a page is basic browser functionality. It is for example there for users who casually need to increase the font size, but you could also use it to quickly check for page layout issues. The keyboard shortcut is the same across (desktop) browsers: Use CMD + on Mac (or CTRL + on Windows) to zoom in. Use CMD/CTRL - to reverse and CMD/CTRL 0 to reset the zoom level. While doing this, observe the UI of your application. All elements should scale proportionally. If this is not the case, the outliers may point you to an implementation issue.

Do not rely on the private tab

It is a good idea to do your tests in a private or incognito tab: This frees you from having to manually delete cookies and clean browser cache before every test. Plus, it switches off all browser extensions that might interfere with your application. However, do not forget to take the real world into account. Nowadays, many people are using extensions that are blocking ads or web analytics requests (popular examples are Adblock Plus and Ghostery). You might be interested to see how your application behaves with such extensions. Apart from obvious effects (ads are not displayed, web analytics is not working), they might as well introduce unwanted side effects, like editorial content not being displayed, web fonts not being loaded, or pages being displayed with visual defects.

OS/hardware/device

Change device orientation

If you are testing on a device that supports device rotation, always check whether this works properly. It is also worth rotating the device in the middle of a use case, regardless of whether the use case is small or large. There are some devices out there (like the Google/Asus Nexus 7 [6]) with a screen aspect ratio that leads most RWD applications to display the handheld view in portrait orientation and the tablet view in landscape orientation. You definitely want to get your hands on one of these devices.

Resize window

There are a several ways to change the browser window size. On iOS devices, you can adjust your browser to share the screen with another app. On desktop devices, you can snap the window to half the screen size or just maximize to full screen with one click. And there is of course that classic way of window resizing: Just by dragging the edge or corner of the window. This one is an interesting gesture with regards to this heuristic. It is because, while doing this, the application has to continually re-calculate what to display. You might observe some issues when doing this in your application (e.g. elements flickering or jumping).

Check at various DPR

DPR stands for "device pixel ratio" [7], which is the ratio between physical pixels and CSS pixels. Next to the viewport size, this is the second parameter determining the UI behaviour of a typical RWD application. But DPR not only affects the UI behaviour: If your application makes use of responsive images, it lets the browser display different image files depending on the DPR. To give a simple example, it is possible to define one image file per DPR, let's say a 100x100 image for DPR 1, a 200x200 image for DPR 2 and a 300x300 image for DPR 3. A classic desktop display with DPR 1 would display the 100x100 image. An iPhone 7 (DPR 2) would display the 200x200 image - and an LG Nexus 5 (DPR 3) would display the 300x300 image.

I would recommend having one screen per common DPR at hand. Take a look at your application on these screens, especially at the images that are displayed. It can be hard or impossible to spot the difference between DPR 1, 2, 3 and 4 images. In such cases, inspect your application in the browser's developer tools. The network tab lists the image files that have actually been requested by the browser.

Some examples for devices with common DPR values:

- DPR 1 - Common desktop display
- DPR 2 - iPhone 7, iPad Air 2, MacBook Pro with Retina display
- DPR 3 - LG Nexus 5
- DPR 4 - Samsung Galaxy S7

It is also possible to simulate DPR 1-3 in Chrome Developer Tools, which works really good. Or, if you have a Retina MacBook and an external DPR 1 display at hand, you can easily switch between DPR 1 and 2. Keep in mind that you always have to reload your application after having switched to another DPR.

With and without scroll bars

You always should check your application with and without scroll bars – and here is why: Some operating systems display scroll bars, some do not. Others, like macOS, leave that decision to the user. And even more, depending on the OS, a browser may or may not take scroll bars into account

when calculating the viewport size. So, the application renders slightly different across operating systems.

When it comes to the application itself, you may for example observe a scroll bar suddenly appearing because more content is being displayed on a page - causing your UI to flicker or jump. This would not be an error though. In some cases, however, scroll bars can indicate issues in your app: For example, if there is more than one scroll bar displayed. Or if a scroll bar indicates a really large page while the page content is actually short. Or if a horizontal scroll bar appears but it should not.

For a quick test, there are several options: If you are on a Mac, you could simply switch scroll bars on and off in the system preferences. If you are on a Windows machine, you can switch scroll bars on and off in the Developer Tools of Chrome by setting the device type either to "Desktop" (scroll bars) or "Mobile" (no scroll bars).

Change input device

Starting with the era of "personal computing", the most common input devices have probably been the keyboard and the mouse as a pointing device. Complemented by the touch screen during recent years, these can be considered most popular. But there are others you might not think of immediately, like TV remote controls. Also, pointing devices are available in several flavours: Mice with or without scroll wheels, trackpads, trackballs, styluses and Apple Pencils etc.

At least have a wheel mouse, a trackpad and a touch screen at hand. The touch screen can be of course part of a mobile device. Then go and check that all the touch and mouse gestures are translated into the desired actions. For example, it could be that scrolling through your application works perfectly fine when using it on a MacBook with trackpad, while scrolling with a wheel mouse takes forever. Just blow the dust off that cheap wheel mouse, plug it in and check it. And last but not least, do not forget about the keyboard! People with disabilities may depend on the keyboard. Not to forget about many others who use the keyboard to speed up their work.

If you lack a touch device or want to do a quick check, you can simulate touch events in the Chrome Developer Tools. However, keep in mind that this is by no means 100% comparable to a physical touch screen, so it is a good idea to always have at least one touch screen device available.

Check at the lower end

Many people in developed countries own quite current mobile devices. However, keep in mind that some do not follow the typical 2-yearly renewal cadence. Others simply just buy used devices and they might still have a current OS version on it. For example, at time of writing this, an iPhone 5s (released 2013) is still compatible with iOS 11 (released 2017). Another topic is screen size. Despite the fact that mobile device displays tend to get bigger, Apple for example still offers the iPhone SE with a 320px wide screen.

Just grab the oldest device you can get that is OS-wise still supported by your application. Does your application feel "smooth" on that device? Observe the UI when scrolling and switching

between portrait and landscape. Observe animations, if there are any. Observe the time between your input and the response of your app.

Throttle/cut connection

Network connection is of course essential for a web application. Keep in mind that your application will be consumed by users on the go with rather slow connections. So, you definitely want to try out how your application behaves with both a slow connection and no connection. By throttling it you are able to properly see whether some kind of loading indicator is implemented and if it works okay. By switching off the connection you are able to check how robust your application is in case a request cannot be made, e.g. whether it displays a proper error message in this case.

For a quick test, Chrome Developer Tools allow for easily simulating a slow or unavailable network connection. If you like to test on a throttled connection in other browsers and on mobile devices, use a proxy application like Charles [8]. Charles also comes in handy when you want to view the network traffic across browsers and devices.

Application

Localization

If your application is being localized, try to get all the translations as early in the implementation stage as possible and have them displayed in your application. This is because you would want to test several things: Do all the copies for headlines, navigation items and buttons fit? Do they even fit on a small screen (for example, French copy is known to be longer than English copy)? Do your fonts contain all the characters necessary?

In case the translations are not there yet, consider mocking these up by using dummy text generators like lipsum.com. [9]

Examine z-indexed elements

The z-index CSS property [10] determines how overlapping elements are ordered on the z-axis. This is basically defined by setting an integer number for the desired element. There's no mechanism in place that provides guidance in assigning the right value or that prevents assigning the same value to several elements. And even if you never assign the same value twice, there's the chance to introduce unwanted behaviour – because it's hard to keep an overview of all the already existing elements and their z-index values. So, if you are building a web application from scratch and you need to use z-index, you will most probably run into issues. This is why I decided to include z-index as a dedicated topic here, even though it is not directly related to RWD.

What can go wrong? Let's start with the easy one first: It might not work at all, i.e. elements meant to overlay others just do not overlay them. Or background elements might bleed through an overlaid element (this often happens when your application features a sticky header). A problem revolving around modal windows is "scroll-bleed". It means that scrolling through a modal does not work, instead of scrolling the modal content, the background is scrolled. You might also experience

that, when having clicked an element inside a modal, the click is actually being received by an element in the background.

Check HTML <head>

The <head> element [11] is a central piece of your application, however many things defined there are as such not directly visible for a user of your application. For example, it provides important configuration such as character encoding, rendering behaviour and references to CSS and scripts. Not to forget about the page title, description and keywords. Furthermore, it can hold references to a plethora of icons (the favicon is only the simplest one possible) and information to be displayed on social networks like Facebook and Twitter when an URL of your application is being shared there. Whether your application is "mobile-web-app-capable" is determined by tags inside the <head> element as well.

To uncover possible issues, you can just take a look at the HTML file of your application and see what is missing there. Page title, the viewport tag and character encoding should be defined as an absolute minimum (plus description and keywords for search engines). In case there are no references to CSS, there is a chance that your application or parts of it will be rendered without any custom styles applied. Script references nowadays are often contained at the end of the <body> content, so missing script references in the <head> element do not necessarily indicate a problem. If your application supports versions of Internet Explorer prior to 11, you should consider having a <meta http-equiv="X-UA-Compatible" content="IE=Edge"> element inside your <head> element. It is also recommended to have a favicon, but it has not necessarily to be referenced in the <head>. You are easily able to spot a missing favicon by just looking at the current tab of your desktop browser. Anyhow, that whole favicon topic got a bit more complicated over the last years, so let's take a closer look.

The classic minimum requirement for a favicon is still valid. You could just put a 16x16 favicon.ico file in the root folder of your application (or define another folder in the <head>) and you are done. However, this would not cater for many use cases: Internet Explorer needs a separate definition for bookmarks. iOS needs separate files in various resolutions that are used when users bookmark a URL on their home screen. This goes in a similar manner for Android. Safari on macOS needs a separate file for pinned tabs. And Windows 8 and 10 need another one that is being used when users add a shortcut for your application to the start menu. For the complete picture, please take a look at this excellent Favicon Checker. [12]

Regarding meta information for social networks, it is also quite hard to get it right. There are different formats enhancing defined by Facebook and Twitter, which not only require texts, but also images in certain dimensions. So, it could easily happen that you have all elements defined, but for example the image has the wrong dimensions. And once you shared your application URL, the shared information might be cached really long (at least this is the case with Facebook). That means that even if you applied necessary changes, they might not be immediately visible on Facebook - and this can be really confusing. It is therefore recommended to use the Facebook Sharing Debugger [13] - it not only allows for checking the content of your meta information, but also lets you invalidate the information Facebook caches about your application.

A lesser-known metatag is the Apple-specific "apple-mobile-web-app-capable" (and the Android-specific equivalent "mobile-web-app-capable"). If it is configured properly, you can try it out like so: Open your application in Safari and use the "Add to home screen" functionality. This creates a shortcut icon to the application URL on the home screen. If you then use that shortcut, your application will open in the browser - but without all the browser chrome like address bar and back/forward buttons. So, if your application is mobile-web-app-capable, you should check whether your application could still be used without those back/forward buttons provided by the browser.

That has been a lot of information. For a quick check, always have a look at the <head> element in the HTML file and at the current browser tab. For checking all that advanced favicon fancy, you could use the Favicon Checker [12] mentioned above (the same page also provides a favicon generator). To check whether there is valid meta information for social networks and to preview it, Facebook and Twitter provide the right tools (Facebook Sharing Debugger [13], Twitter Card Validator [14]). Keep in mind that these only work with publicly available URLs.

Observe the network traffic

Your application most probably references to other resources that are requested as soon as your application URL is being opened in the browser. And there might happen further requests to more resources while your application is being used. These can be styles, scripts, fonts, images, web analytics requests etc.

It can happen that downloading or requesting these resources does not work and you may not notice that by just looking at the UI of your application. Good examples are server requests generated by a web analytics solution, which you can only see in the network traffic. So, you should always keep an eye on the network tab of your browser. Also, you could try blocking certain requests to see how robust your application is. For example, block images to see whether a placeholder is displayed. Or block fonts to see how your application looks with the fallback font being rendered.

While checking the network tab, you would also want to look for things like duplicate requests, unwanted redirects, requests that run into errors (like 500, 403, 404, ...) and timeouts, un-minified Javascript/CSS files and responsive images not being loaded correctly (e.g. the bigger file size is requested). Also, keep in mind that your application will be consumed by users that are on the go with a rather slow connection and a mobile data plan. So, watch out for single big files and the overall application size. And keep an eye on the number of requests being made.

The Chrome Developer Tools network tab [15] has a very nice summary for the overall number of requests, overall file size and load time. When it comes to testing web analytics, the developer tools of all modern desktop browsers do a good job in displaying all parameters appended to a web analytics request URL in a human-friendly format. And for blocking requests, you could use Chrome Developer Tools or a proxy like Charles.

Look closely at the font

Not a long time ago, web pages were using pre-installed fonts to render. Headlines were maybe displayed as images. Then came web fonts, enabling a whole lot of design possibilities. Many web applications nowadays are using web fonts. A good practice is to define a fallback font, which is used in case the primary font cannot be displayed.

It is sometimes hard to spot the differences between primary font and fallback fonts. So, make yourself familiar with the differences and details and if in doubt, ask a designer from your team. OS-wise, some older Android versions often have problems with loading web fonts. It could also be that certain characters are missing in the font, so you would want to test for that in at least the languages your application supports. It could also be that only single parts of your application display the primary font, while others do not. You should also check that the fallback scenario works: Block the primary font from being loaded and see whether the texts in your application still look acceptable and how your application UI behaves.

[1] - https://en.wikipedia.org/wiki/Responsive_web_design
[2] - Chrome: https://developers.google.com/web/tools/chrome-devtools/device-mode/emulate-mobile-viewports#responsive_mode
[2] - Firefox: https://developer.mozilla.org/en-US/docs/Tools/Responsive_Design_Mode
[2] - Safari: https://support.apple.com/kb/PH26266?viewlocale=en_US&locale=en_US
[3] - https://developers.google.com/web/tools/chrome-devtools/device-mode/emulate-mobile-viewports#media-queries
[4] - http://farukat.es/journal/2011/02/528-modern-browser/
[5] - https://www.smashingmagazine.com/2016/02/high-impact-minimal-effort-cross-browser-testing/
[6] - https://en.wikipedia.org/wiki/Nexus_7_(2013)
[7] - https://developer.mozilla.org/en/docs/Web/API/Window/devicePixelRatio
[8] - https://www.charlesproxy.com/
[9] - http://lipsum.com/
[10] - https://developer.mozilla.org/en-US/docs/Web/CSS/z-index
[11] - https://developer.mozilla.org/en-US/docs/Learn/HTML/Introduction_to_HTML/The_head_metadata_in_HTML
[12] - https://realfavicongenerator.net/favicon_checker
[13] - https://developers.facebook.com/tools/debug/sharing/
[14] - https://cards-dev.twitter.com/validator
[15] - https://developers.google.com/web/tools/chrome-devtools/network-performance/reference

Marcus Noll - Senior Software Engineer, XING SE, Hamburg, Germany

Currently adding the tester mind-set to the Frontend Architecture Team, which is building and establishing a Web Design System based on React. Before that, 4+ years of QA engineer experience in the field of responsive web design, building and leading a small QA team. Thrilled by the fast-paced world of web frontend development. Father of two.

Marcus all kinds of weird music, but also has a big heart for certain mainstream stuff.

Contact Marcus via XING - https://www.xing.com/profile/Marcus_Noll3/cv

Building QA Processes

- by Tiago Correia

Some context

I began my journey at Talkdesk in October 2016, motivated by the big challenge of building a QA process, almost from scratch. Since day one I could see that all my colleagues were wondering how I was going to tackle this challenge and implement a QA process. At the time I joined the Engineering team, its size was around 60 people, organized in scrum teams of 5 or 6 developers, and there was an independent QA team with 2 testers.

Before starting to even think about making any changes I wanted to get to know the people I was going to work with: their profiles, their expectations and their concerns. The QA team didn't have much knowledge of QA processes and best practices, but they had a deep understanding about our product, which was key on my ramp up. Another interesting outcome was that both testers showed concerns regarding not having time for exploratory testing.

While getting to know the team and the processes used I could clearly see the lack of a QA process, how the QA team was overloaded with work, but I also noticed some worrying practices. The one that alarmed me the most was when developers wanted a tester to test a given User Story, they would sit together and the developer would give information about the feature and almost drive the testing process because the tester didn't have sufficient knowledge on the new feature. I could see straight away that the testers needed to be involved earlier on in the process, having information about the new features in advance in order to prepare and properly test the story. What value does a tester add when he is only performing the test cases a developer asks for? It was clear to me that we needed to change mindsets in order to implement a solid QA process.

Finding Test Management Tool

It came as no surprise to me that there was little documentation regarding the testing process and test cases. There were a couple of "how to" wiki pages, a few wiki pages with bullet points with features that needed to be verified in smoke tests[1] after deploys and also some pages listing a few test cases for the core functionalities. Despite scarce, this documentation would prove to be key in the near future.

In order to start having better documentation and, more importantly, to give more structure to the process we were building, we felt the need for a Test Management Tool (TMT). This new necessity triggered some thoughts: How can a TMT fit our business? What are the biggest pain points when using this type of software? What are the main characteristics I need to take into account when choosing a TMT? To answer these questions I mapped the requirements I thought were relevant on a TMT:

- Integration with our project management tool - Allowing a quick relationship between User Story and Test Cases.

- Usability - "Easy to use", the team wasn't much experienced using these tools, and that would make adoption easier.

- Test cases re-usage - One of the pains I felt when using these tools was that I often couldn't reuse the same test cases in different context. A tool which allow test cases reutilization would potentially allow us to gain time when specifying test for new features.

- Customization level - More customization usually means more complexity, but we needed the tool to be adaptable to the process and the business.

- Continuous Integration (CI) and Continuous Deployment (CD) - Thinking on a future with automation testing, the CI and CD pipelines would feed test run results into the tool.

- Price - Always a very important factor. We scheduled demos, watched videos, tried a few trial versions and created lists with the pros and cons of each software in order to evaluate the different options. After carefully analyzing and comparing all the alternatives, we made our choice.

[1] Smoke tests are a set of tests per application, performed in production right after a deploy, to assess core functionalities. The aim is to run the smoke tests suit in less than 30 minutes.

Assigning testers to teams

I could see developers were seeking the testers' help more frequently, especially at the end of sprints. We were only 3 testers and it was difficult to attend all the testing needs of each team. The testing effort was unbalanced, concentrated at the end of the sprint, overloading the tests. If you add to this situation the fact we were only getting information about the stories in the last few days of the sprint, testing was becoming very hard.

I saw this as an opportunity to challenge the testers and teams to work more closely. I talked to both sides and showed them all the advantages and the impact we could make if the testers were more involved in the teams' daily work, being part of the team and participating on the different sprint ceremonies. Getting testers involved in the development process from the beginning is crucial for the testers to do their job, understand the requirements sooner, helping define acceptance criteria, think about the testing scenarios and consequently finding issues sooner.

The proposal was well received by all, testers and development teams. Each tester was assigned to two frontend teams according to their areas of expertise and I was assigned to the remaining 3 front-end teams. These were hard times for me: my calendar was completely packed with overlapping meetings for 3 different teams and I didn't have time to properly test and help each team. I was feeling frustrated, overloaded, but, with limited resources, that was the only way to show to everyone that this was the right process move and to improve the quality of the product.

Scaling the team

Being part of the development teams made our days as testers even busier and more teams were requesting QA resources and time. It was symptomatic; we needed to increase the number of testers in the company. The team and the process were stepping up and I wanted to grow the team in a stable and sustainable way. Hiring a lot of people would be hard to justify to the management team, and the more people you hire, the more instability you introduce to the team and the process. Therefore, we decided that we weren't going to hire that many testers, but we had to look for the right profiles and define exactly what we were looking for in a tester.

The testing team had very good product knowledge, but lacked some technical QA expertise, which was making it very demanding and time consuming for me as the only senior tester in the team. It was time to find another senior tester who could boost the team with their experience and expertise. We looked for this profile and added 2 testers to the team. At Talkdesk, things tend to happen very quickly. The decision to open our Porto office was seemed taken almost overnight. With the engineering team growing faster in both locations, Porto and Lisbon, it was with no surprise that we started looking for possible candidates for our QA team in Porto and ended up hiring 2 more senior testers.

Consolidating the Process

Once we got the licenses to use our Test Management Tool, it was clear to me the tool had to be introduced gradually. I put together training sessions with the testers, where I presented the software, and we also discussed our regression and smoke testing initial batches. The discussion resulted in documentation prioritizing and cataloging the tests, which we later imported into the tool.

Testers were now able to document their work more consistently. During each team's grooming session, the testers analyzed the next stories in order to have enough information to specify the test cases. Once the stories were developed, the testers would run the manual test and log the test execution data. All of this work was not reflected in the teams' sprints, because the testers didn't

contribute for the story's estimation points therefore, we could be a hidden bottleneck. I knew this was one of the reasons why sometimes teams didn't deliver all the stories and I needed to do something to improve it. I arranged meetings with some of the team leads and explained to them why I thought testers should participate on estimations and we agreed I should start estimating user stories. The change was well received by everyone and produced good outcomes. A couple of weeks after, I proposed the same approach to the other teams and all the testers started to be involved in the estimation of stories.

Over the last few months, all testers have been struggling with time. We got together to analyze where we were using our time. Dealing with two teams and the constant context switching, smaller and more frequent deploys to Production meant more frequent smoke tests and many more tests cases to specify. Hiring more people to allow each tester to focus only on one team was not an option, so to address these time constraints issues we decided to create a guideline for test case specification, granting the same test specification principles to everyone. We also prioritized testing activities, enforcing the importance of the hands-on testing and minimizing the sprint story life cycle.

Challenges ahead

Today we have a stable manual QA process and, like any other team, there is much room for improvement. In our QA team, I see two main areas where we should primarily invest. The first one is a classic in modern times: Automation. We need to automate our process as much as we can, and by this I mean not only automating our testing but also our continuous deployment. In the past, we've made some attempts to introduce UI automation tests using Selenium based approaches, but we suspended those, as the results were not consistent and ended up with too many false positives and false negatives. Our business is very prone to this kind of UI testing results, so I believe we should go for a very strong base of unit and integration testing layers and a small and simple UI automation layer. By having a strong automation process included in our continuous deployment pipelines, it will free up our testers from their daily manual validation tasks after each deploy. The second area we need to invest in is people. We want to have one tester per team, enabling the tester to perform all of the QA tasks they currently struggle with due to time constraints. Each tester will be focused on a specific team and will be able to specify test cases before the development phase, enriching the feature and anticipating possible issues. Having the developers aware of the test cases in previous stages will also speed up test cases automation and allow the testers to perform exploratory testing approaches covering edge cases and paths that haven't been considered.

By having testers involved in the automation work, whether it is defining test cases, developing the tests or adding them to deployment pipelines, the quality of each release will increase.

Tiago Correia is QA Team lead with nine years experience plus two as a developer. He has strong knowledge about QA processes and methodologies including mobile and web platforms.

In his professional career he has worked for organizations such as Blip (Betfair), Farfetch and he nowadays works at Talkdesk.

Sharing is caring! Don't be selfish ☺

Share this issue with your friends and colleagues!

Happiness is….

Taking a break and reading about **testing**!!!

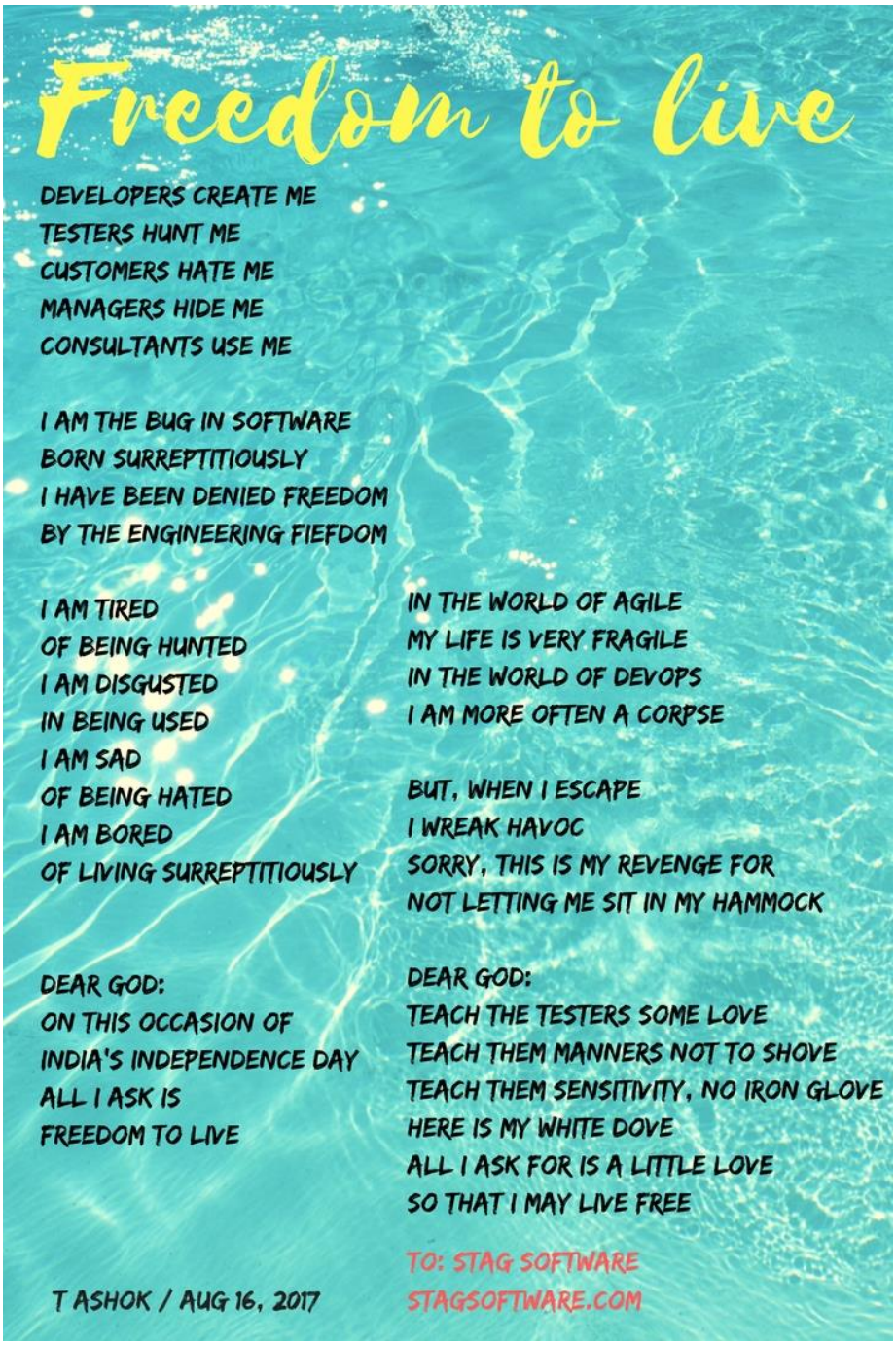
Like our FACEBOOK page for more of such happiness

https://www.facebook.com/TtimewidTesters

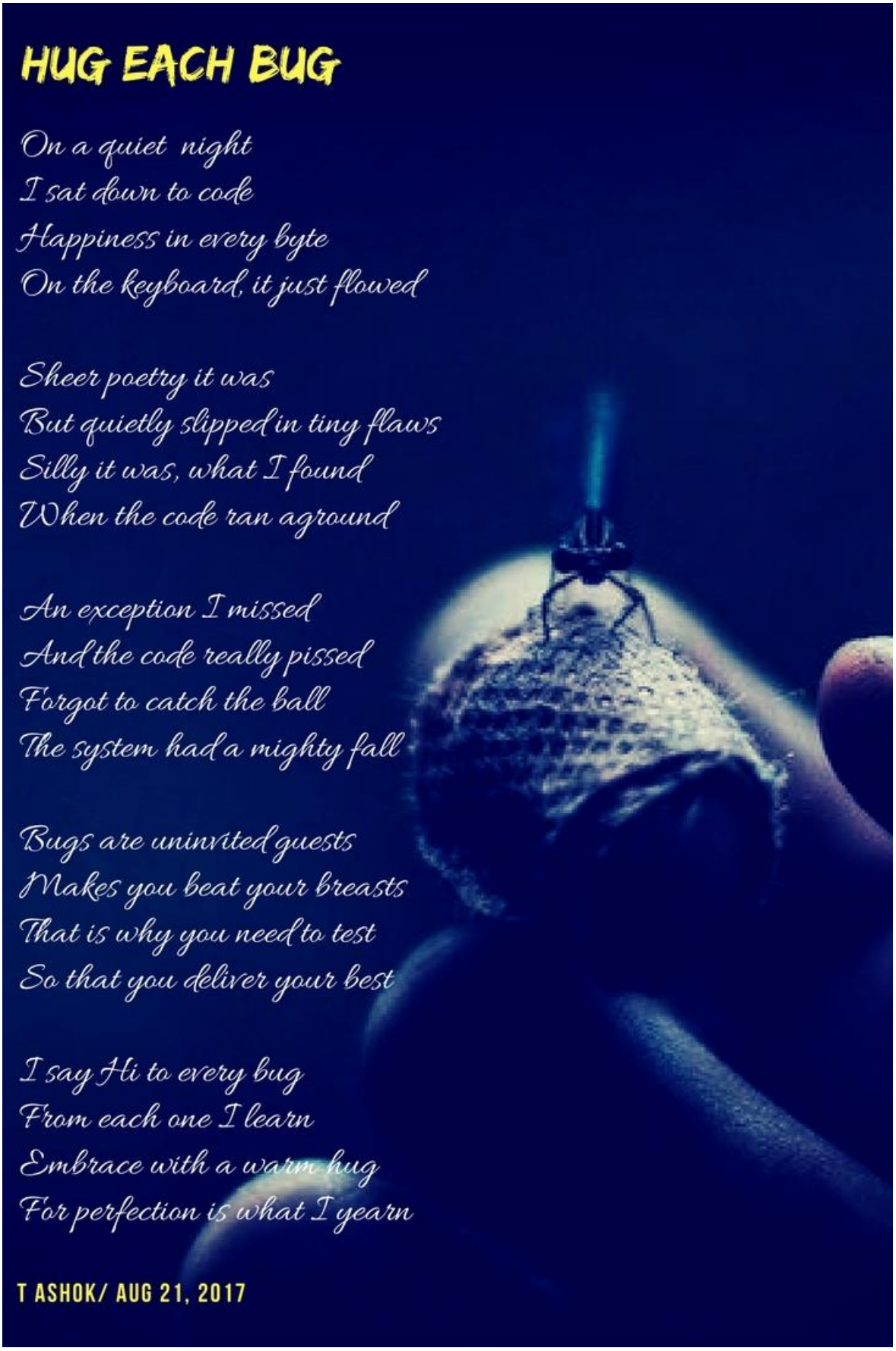
Three Poems on Testing

This is a very different article that consists of three poems on testing written from the viewpoint of tester, developer and manager. Hope you enjoy the light hearted humor!

Here is a poem “Freedom to live” where a bug begs for freedom to live. Written on India’s independence day (Aug 15), it is a plea to God to let him live. This is sent as a note to STAG Software, a software test boutique. This is written from the tester’s viewpoint.



The next poem "Hug each bug" is written from the view of a developer who learns from each bug to write superior code.


HUG EACH BUG
On a quiet night
I sat down to code
Happiness in every byte
On the keyboard, it just flowed
Sheer poetry it was
But quietly slipped in tiny flaws
Silly it was, what I found
When the code ran aground
An exception I missed
And the code really pissed
Forgot to catch the ball
The system had a mighty fall
Bugs are uninvited guests
Makes you beat your breasts
That is why you need to test
So that you deliver your best
I say Hi to every bug
From each one I learn
Embrace with a warm hug
For perfection is what I yearn
T ASHOK/ AUG 21, 2017

The third poem "I am a manager, I have to execute" is a serenity prayer from a manager to handle projects that become hot due to bugs!

T Ashok is the Founder &CEO of STAG Software Private Limited.

Passionate about excelle
mission is to invent techn
deliver "clean software".

He can be reached at
ash@stagsoftware.com

What I learned about testing from Stoicism, Epicureanism, and Andy Zaltsman

If you've never heard about Andy Zaltsman, and if you enjoy sarcastic political humor, I recommend you listen to The Bugle podcast.

I am mentioning him because he recorded a 3-episode series with the BBC2 around the teachings of the Greek philosophies and how this would apply to today's life called "My Life As…"

Some 25 years ago, when I was in college back in Costa Rica, I studied one or two courses on philosophy and we talked about the classical schools, but listening to the series I realized I had already forgotten everything I managed to learn back then.

But more interesting to me was that, as I listened to the podcast while stuck in traffic, I could not wonder to see a number of things we could apply to our testing from at least 2 of the schools he reviewed on the series: Stoicism and Epicureanism.

There was a third episode about Cynicism that gave me a number of good personal tools, but less thin to apply to testing, and so I am leaving it out for now

Let's dive in and understand some of the stuff we can apply to our testing from these two philosophical schools.

Stoicism

Stoicism believes that via self-control and individual development (and self-understanding) we can overcome most of the challenges in life.

> One of the stoic teachings that resonated with me most was that "***We are not disturbed by events, but by our beliefs (or perspectives) of the events themselves***".

Once you think about the sentence it starts making sense, but I also think it is also easier said than done.

From the stoic ideas brought forward on the BBC series I thought interesting a couple of exercises or practices that can be taken from the classical stoics:

1. **Clearly define your intentions for every day –** At the beginning of your workday, take some time to clearly define your intentions of the day. Write down a short list of tasks or actions you want to do, and review this list once or twice during your day to make sure you have not been derailed by the daily rollercoaster of new tasks and other distractions that bombard us constantly.

2. **Finish your day by running a balance your achievements and losses –**

Either at the end of your workday or at night before you go to bed, take some quiet time to recollect all your achievement and your challenges of the day. Look at all the good things and the bad things in equal light, concentrating on how not to run into the same mistakes in the future. Very much like a agile retrospective, but daily and personal.

In a more general perspective, the thought that sums Stoicism up for me, is that I am responsible only for those things that I can control, and the things that I cannot control I am not responsible for. Based on this understanding I should concentrate completely on attending those aspects under my control, while not losing too much sleep over those that are beyond me.

This resounded with me deeply and showed me some clear mistakes I made back when I was starting testing. I remember especially how I used to suffer for finding really big bugs in the project; especially bugs that I knew would delay the product launch date. I always felt half responsible for the bugs and for what they would do the project and to the developers who had written them. Sounds weird? Sure, today it does, but back then it was part of my daily experience as a tester working in a startup company that was desperately trying to close its initial deals.

My "today" tester would really like to go back in time and tell "young Joel" that it is foolish to feel responsible for the bug he found. That the bug was there initially, and finding this bug is better than letting the end user find it. What is my responsibility is not writing or fixing the bug, only finding it in an efficient and timely manner.

Epicureanism

Many of us wrongly believe that Epicureanism is about the pursue of unmeasured enjoyment, large feasts, illogical luxury – well maybe not many of us, but at least those of us who think about this topic at all. While in truth, it could not be further from the truth.

It is correct that this school of philosophy believes the aim of life is the individual pleasure, but it also teaches to find pleasure in the simple and little things in life.

There are a number of things to learn from Epicureanism, but the part that I took into my "testing life" is the part where it tasks us with understanding more deeply which of our needs are wants are really important and separate them from the ones that are "empty" or unimportant.

Audit yourself – you should do a self-assessment to understand what we want to do and what we need to do. Which are the things that are really important to us and those that we are only doing because we believe it is what is expected of us without seeing much sense or internal importance on it. Obviously, once you make this distinction you will know what to keep doing and what to stop doing – hopefully.

Write your obituary or your retirement speech – a tool I found extremely concrete is the exercise of writing your own obituary, or if you find it is easier you can write the speech someone will give about you on your retirement party.

The idea is to think about what attributes or actions you will want others to remember you by, and this will be a good indicator of the areas where you need to invest as a person or a professional.

Looking for teachings in other places

One of the things that writing this blog post reminded me, was of questions we have in the State of Testing Survey that was originally proposed by Jerry Weinberg. The question is about how do you learn about testing as part of your career, and the addition Jerry added to the list of possible answers was that we learn about testing from other fields we are in contact with as part of work and of our daily lives.

This was for me a great of example of simple things in testing we can learn from the teachings of the Greek philosophers from thousands of years ago!

Joel Montvelisky is a tester and test manager with over 15 years of experience in the field.

He's worked in companies ranging from small Internet Start-Ups and all the way to large multinational corporations, including Mercury Interactive (currently HP Software) where he managed the QA for TestDirector/Quality Center, QTP, WinRunner, and additional products in the Testing Area.

Today Joel is the Solution and Methodology Architect at PractiTest, a new Lightweight Enterprise Test Management Platform.

He also imparts short training and consulting sessions, and is one of the chief editors of ThinkTesting - a Hebrew Testing Magazine.

Joel publishes a blog under - http://qablog.practitest.com and regularly tweets as joelmonte

Advertise with us

Connect with the audience that MATTER!

Adverts help mostly when they are noticed by **decision makers** in the industry.

Along with thousands of awesome testers, Tea-time with Testers is read and contributed by Senior Test Managers, Delivery Heads, Programme Managers, Global Heads, CEOs, CTOs, Solution Architects and Test Consultants.

Want to know what people holding above positions have to say about us?

Well, hear directly from them.

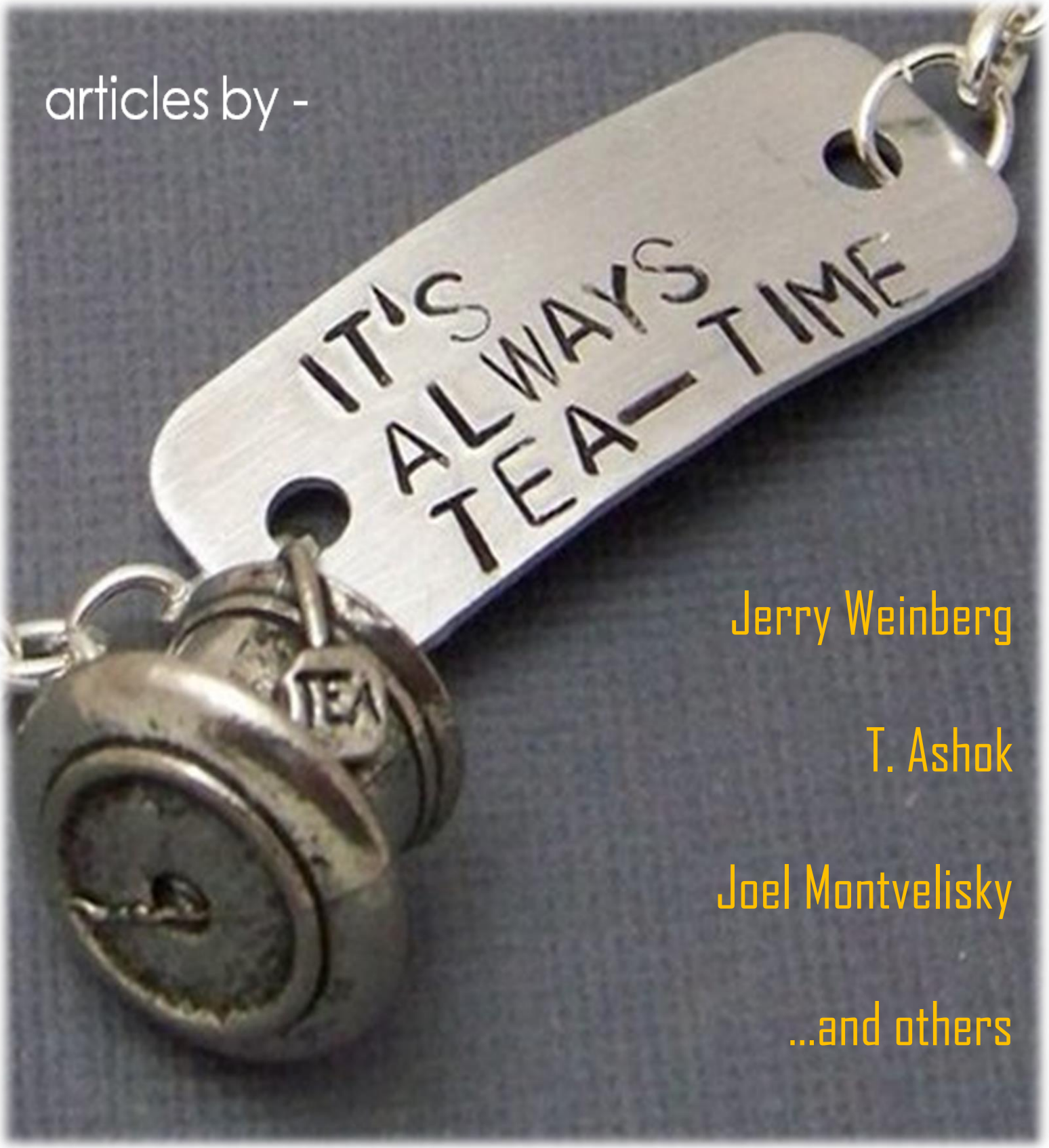
And the **Good News** is…

Now we have some more awesome offerings at pretty affordable prices.

Contact us at sales@teatimewithtesters.com to know more.


Every Tester
who reads Tea-time with Testers,
Recommends it to friends and colleagues .
What About You ?
Image : vernhart

in next issue



our family

Founder & Editor:

Lalitkumar Bhamare (Pune, India)

Pratikkumar Patel (Mumbai, India)


Lalitkumar


Pratikkumar

Contribution and Guidance:

Jerry Weinberg (U.S.A.)

T Ashok (India)

Joel Montvelisky (Israel)


Jerry


T Ashok


Joel

Editorial | Magazine Design | Logo Design | Web Design:

Lalitkumar Bhamare

Cover page image – Metaphrasi

Core Team:

Dr.Meeta Prakash (Bangalore, India)

Dirk Meißner (Hamburg, Germany)


Dr. Meeta Prakash


Dirk Meißner

Online Collaboration:

Shweta Daiv (Pune, India)


Shweta

Tech -Team:

Chris Philip (Mumbai, India)

Romil Gupta (Pune, India)

Kiran kumar (Mumbai, India)


Kiran Kumar


Chris


Romil

|| Karmanye vadhikaraste ma phaleshu kadachna | Karmaphalehtur bhurma te sangostvakarmani ||

To get a **FREE** copy,

Subscribe to mailing list.

SUBSCRIBE

Join our community on

facebook

Follow us on - @TtimewidTesters

www.teatimewithtesters.com