THE INTERNATIONAL MONTHLY FOR NEXT GENERATION TESTERS

Tea-time with Testers

JULY-AUGUST 2015 | YEAR 5 ISSUE VI

Articles by –

Jerry Weinberg

Huib Schoots

Matt Heusser

Wayne Yaddow

Joel Montvelisky

Rahul Verma

T. Ashok

Over a Cup of Tea with Dr.Cem Kaner

Sign Up For A Free Demo Today

www.teamqualitypro.com

GET A FULL VIEW INTO ALL YOUR DATA

- Inhouse/Offshore contracts
- Defects
- Test phases
- Software Development
- Reports
- Payroll/ Contracts
- All projects
- Notifications

- Project Progress
- Team Analysis
- Top 10 Best / Worst
- Cost Measurement
- Defects / Testing

TAKE THE FIRST STEP IN SAYING:

YES TO:

- Real time reporting
- Instantaneous data gathering
- Objective, vetted data
- Comprehensive data
- Data that mirrors your process
- Data that is always current

NO TO:

- Manual reporting
- Data warehouse projects
- Subjective data
- Missing data
- Not aligned data
- Wrong time frame data

TEA-TIME WITH TESTERS

First Indian testing magazine to reach 115 countries in the world !

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: (+91) 8275562299
Pratik: (+49)15215673149

Editorial

Let's talk testing again

Dear Reader,

While writing the editorial this time, I realized that I have been writing about things that I feel are important. Good thing is, from your letters/comments it seems that most of you are able to relate those ideas with your day to work. However, I feel that it's good to have discussions once in a while.

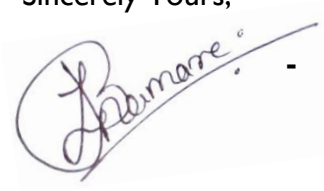
So, no pearls of wisdom from my end this time ;-). Instead, I want to hear ***your*** stories. So, how are you doing? What new is happening in your group or organization? Did you learn any new skill? Any learning from articles that we publish that you implemented at work? Any local group, peer conference that you set up? Attended a conference? Won testing competition? Or…….. missed a bug in production ☺ ?

Well..well..well…I think there is a lot more we all have to share with each other. Why not just send me letter or share your stories on our facebook page? May be we can work together to create a fantastic article based on your story or maybe we all can just confer through discussions and learn from each other. Aha! I find that exciting. Don't you?

On that note, I will take your leave for now. And I will be eagerly waiting to hear your test side stories…

Enjoy another cool edition of Tea-time with Testers. Happy reading!

Sincerely Yours,



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

QuickLook

QAI in association with Edista Testing Institute (ETI) announces the 15th Annual International Software Testing Conference (STC 2015). The conference is scheduled for December 03 - 04, 2015 at Bangalore.

Through the theme, at STC 2015 we propose to explore practices which reflect knowledge in general, intentionality, introspection (and the knowledge it specifically generates) and phenomenal experience. STC 2015 explores ideas for new products, new ways of working, and for new strategies, still others for entirely new lines of business, resulting in inspired innovation and increased agility. As a platform, STC 2015 unravels examples, best practices from organizations and practitioners across the world.

The 15th Annual International Software Testing Conference (STC 2015) will take place on 3-4 December, 2015 in Bangalore, India. Built on the theme of Conscious Quality - Delivering Value Constantly, Consistently, and Continuously in an Agile World, this year the conference will be a display of ideas, experiments and experiences to explore challenges and suggest techniques and Best Practices to successfully engineer quality in a continuous, constant and consistent way.

Know more

STATE OF TESTING 2015

Results of the Largest Worldwide Testing Survey are out!

Are you curious to learn more about the testing community? We believe that all testers are!

The State of Testing Survey-2015 results are finally here and we are sure that you'll love the report when you see it. More than 860 testing professionals participated in this survey which helped us gather a lot of interesting insights on questions around the current state and the future of software testing.

You can download the Survey Report from here → **Download Survey Report**

You can also get free copy of previous survey report – State of Testing Survey 2013

A big thanks to all our contributors

We want to thank all the QA & Testing bloggers as well as the tweeps who have been helping us reach more testers worldwide by sharing this survey and its results with their audience and followers.

State of Testing is brought to you by **Tea-time with Testers** in association with **PractiTest**.

TESTEA TALKS

Evolution of the Art and Craft of Testing cannot be discussed without citing credits to the work done by **Dr. Cem Kaner** in last few decades. Dr. Cem's contribution to this field has had a significant influence on the way testing is being done by thinking and adaptive testers today.

Conversing about testing with Dr. Cem, knowing more about his journey and knowing more about his opinions on various subjects in the field was on our wish list from long time. I got to speak with Dr. Cem on various aspects of his life, expertise, opinion and suggestions recently. And this 'series' is a result of what we have been discussing from quite some time.

In part 1 of interview with Dr. Cem Kaner, you read about his career journey with some interesting stories and lessons learned. We also discussed about his views on getting testing education from commercial training courses and from universities. To catch up, you can read part 1 here. Read on to discover what more we conversed….

- Lalit Bhamare

Over a Cup of Tea with Dr.Cem Kaner

You have experience and interests in multiple fields like Economics, Law, Mathematics, Psychology, Professional Writing, Software, and Teaching. How do you find balance between your interests being involved in them at same time?

I don't see these as multiple fields. The focus of my career has been the safety and satisfaction of software customers and software development workers. This is a complex topic. To have any hope of understanding it, I have to consider the same problems from different viewpoints.

What in your opinion makes a good / better / best tester?

I interpreted this question as focused on recruiting testers—how to choose a good/better/best tester.

From that viewpoint, I think there are many ways to be an excellent tester. The essence of the testing role is to investigate a software product or service in order to discover information about its quality. The tester typically conducts this investigation on behalf of other people. We usually think of testers involved in the process of developing a product. But other investigations use testers too. For example, maybe they're trying to understand what risks come from adding this software to an existing service. Or maybe the testers are investigating the causes of an accident.

The ideal person for this investigation will be able to efficiently find new information that has value to the project, able to explain what was found in a way that works for the people who need the information, and is willing and able to work in a way creates trust, respect, and a pleasant workplace. Many different skills are in play here and they vary by project. For example,

- I've been working with a course management system that has a huge number of features. They put new builds into production every three weeks. Over time, the overall product is improving, but for months, it feels like they are changing an area by swapping new bugs for old ones. If I could hire only one tester for this project, I think the greatest value would come from someone who could improve the regression suite, perhaps with a focus on code-level regression tests to support refactoring.

- Contrast this with software designed to spot patterns in very large data sets and suggest new actions based on those patterns. If I could have only one tester on this project, I would most want someone who understands the underlying mathematics and the underlying domain (for example, someone who understands medical records) and who can apply this knowledge to build complex, realistic tests of the validity of the implementation.
Most projects have several testers, not just one. The greatest value from an additional tester comes from someone who brings skills that aren't yet well enough represented on the project. Similarly, the greatest value an additional tester brings to a test group comes from improving the group's mix of skills so that it can be more effective with the types of projects that it normally tests. I wrote a very long paper (55 pages) on this in 2000. The paper tries to help you consider what skills you need in your group and how to decide whether the person you are interviewing has or can develop those skills. For that paper, please see http://kaner.com/pdfs/JobsRev6.pdf

... CONTINUED ON PAGE 52

Tea & Testing with Jerry Weinberg

Habits for Agile Testing

In addition to her many other talents, my wife, Dani, trains dogs. Her specialty is "problem dogs"—which really means "problem owners."

Rather than retrain the dogs, Dani retrains the owners, who then retrain the dogs. In that sense, her dog work is very much like my program testing work. When people come to me with "problem programs," I work on the people, not the programs. After all, it was the people who made the programs into problems.

Of all the questions dog owners ask Dani, the most frequent is, "When do we start training the puppy?" Unfortunately, few people ask me the corresponding question, "When is the right time to start testing a program?"

What is the right time to start training a puppy? As Dani explains, the question is meaningless because "you start training a puppy the moment you set eyes on it, whether you intend to or not. And the puppy starts training you at the same time.

The same is true of testing programs. You start testing a program the moment you start to think of the possibility of writing a program. Actually, it starts even earlier.

How can program testing start even before you start thinking of writing the program? Think about the dog owners. Their troubles arise from bad habits they've acquired over a lifetime, before they ever set eyes upon their new puppy.

Your good and bad habits will have more influence than any other factor in the success or failure of your program testing efforts. It is no accident that certain programmers and testers perform consistently better than others in the production of well-tested programs. Those who believe it is an accident will learn nothing about program testing until they convince themselves it is possible to learn how to do a better testing job.

What Does Genius Require?

I'm not denying that a measure or basic intelligence is required to be a successful program tester. Even some forms of "genius" might help. But those who subscribe to the "native genius" school of programming should study the lives of geniuses.

For example, Einstein once remarked that he really didn't do anything but take the work of others and put it together in some way that had not been done before. If we are to emulate Einstein and put together the work of others—instead of starting from scratch with each new program—perhaps we can approach the genius level in our testing.

Scientists know that there is honor, not shame, in taking the work of others—probably smarter than ourselves—as the starting point for our work. For the most part, this previous work is available to us in libraries of one form or another.

Libraries may store millions of useful programming ideas, but their very size can make it difficult to know what they contain. In order to make effective use of libraries in program testing, we have to know what kinds of libraries are available, and we have to know what those libraries contain.

On library that every programmer uses is the one contained in the programming language. Even if your program is in binary machine code, you're using a library—the one the machine designers thought useful for programming. It is theoretically possible to design a general purpose computer with but a single instruction—a kind of "conditional branch and subtract." Therefore, any instruction set bigger than that must be considered a library—a way of giving the programmer a set of pre-tested building blocks.

By "higher-level" language, we essentially mean a language with a library that allows programs to be built with fewer building blocks. Fewer building blocks means fewer connections between building blocks—which in turn means fewer places to search for errors. Unfortunately, many programmers are not well acquainted with the contents of their language library.

Here's just one example. A programmer came to me with a program that was mysteriously losing parts of strings. I helped him narrow down the problem to a statement that was trying to replace two characters with asterisks, but when the original string was longer than 10 characters, the excess portion was truncated. When I asked him why he had used the "magic number" 10, he explained that at the time he wrote the statement he didn't think he would have strings longer than that.

He didn't know his language library contained the ability to do substring assignment. Had he known this, he could have written a much simpler and more reliable statement. That approach—because it was a more direct expression of what he wanted to do rather than how he wants it done—would have prevented the bug and other bugs from ever occurring in the first place.

Another genius, Newton, said he could see so far because "I am a midget standing on the shoulders of giants." Dick Hamming once told me the programmers are more like midgets standing on the toes of other midgets. It may be true that our language designers lack the mental stature of a Newton, but that's no reason to stand on their toes by ignoring the hidden libraries they have created for us. You can still see farther standing on shoulders than on toes.

Genius Libraries

But how can you come to know the contents of your hidden library? The obvious answer is to read the manual. Ugh! We complain that manuals are badly written, and many are. But a more serious problem is that manuals are written for reference, not to be read like novels. If you know there is such a thing as substring assignment, and you know its name, you can look it up and learn how it works. But if you don't suspect such a function exists, how can you find it in the manual?

Another problem with manual reading is their emphasis on how things work, rather than why you might want to use them in the first place. They are solution-oriented, rather than problem-oriented, leaving their readers to bridge the gap between what they're trying to accomplish and how the language could make it easier and more accurate.

Still, there are ways of using manuals to overcome some of this shortcomings. Two or more programmers can play a game. One takes the manual and reads the name of some feature to the others, who must then describe how that feature works. "What happens when we execute this?" Everyone writes down an answer, after which the example is executed and the competitive members of the team can award points for correct answers.

This game increases everyone's familiarity with parts of the hidden library—or any application, but still leaves the "why" largely unanswered. The game can be extended to "why?" by asking each player to come up with a problem for which his feature is the best solution. For substring assignment, a problem might be "to insert asterisks in the third and fourth positions of a string," or "to replace a blank at position N with a comma."

The technical review is even more effective at vocabulary building. Technical reviews have the further advantage that managers who might be troubled to see programmers and testers "playing games" have little trouble accepting a technical review for its obvious function of finding errors. In my experience, however, the long-range error-prevention benefits of technical reviews far outweigh their short-range debugging efforts.

When several people sit down to review code, each person can quietly witness all the good techniques used in that code. They can also witness poor techniques, and if anyone in the group happens to know a better technique, all can learn it. Moreover they can learn these things without having to confess that they didn't know them before the review.

Another problem with manuals is that though they may be comprehensive, they tend to put the emphasis on what is most difficult to explain, rather than what is most useful to know. Because technical reviews are focused on current work, the learning from reviews tend to be more relevant than what you can learn from playing games.

There are, of course, other types of program library besides the hidden library of the programming language. There are procedure libraries, subroutine libraries, libraries containing tables of common data,

utility libraries, libraries of parameters to direct those utilities in common tasks, macro libraries containing common code sequences, data dictionaries, and the inventory of existing applications—often a huge library of useful code for reuse.

The amount of useful, pre-tested material in such libraries is so overwhelming that programmers and testers sometimes forget about that other sort of library—the kind containing good old-fashioned books. A caution, though. Even if we could somehow guarantee perfect copies of the author's examples, we must remember that examples presented in a book are ordinarily designed—or should be—for maximum teaching effect, not for maximum generality or performance. Books are terrific ways to obtain new ideas of what be in other types of library, which is where you should look for reliable pieces of code and design.

The subject of early testing would be incomplete without the mention of one more type of library—the library of work habits you carry around in your head, or wherever it is that work habits are carried. For instance, have you ever noticed how pig-pen workers invariably spend more time debugging than those with tidier habits? I say this not as Mr. Clean, but as an old pig-pen programmer myself. It took me thirty years of effort to clean up my act reasonably well, but I've never completely eliminated slovenly habits of my youth. Still, little by little, I've eliminated slovenly, lazy practices that had previously wasted thousands of hours of precious time.

Seven Simple Habits

The Chinese say, "a journey of 10,000 miles starts with a single step," and even one bug prevented is a step in the right direction of more reliable software. So, here are a few of the pre-programmed habits I've noticed in effective program testers:

Keeping a Journal

Like all professional engineers, professional programmers and testers should keep a bound journal in which to record all those tiny but significant events that occur every day. (I prefer a bound journal to a digital record because it prevents me from revising history to make myself look better than I really was.)

Ideas, observations, events, successes, failures, puzzles—all are grist for the journal. You don't have to realize the importance of something at the time you write it down. Indeed, the puzzling things are the most important to record. Often, bugs are found only when they have managed to repeat themselves in a dozen ways, and only with a journal will you have all the necessary data in one place. Memory isn't sufficiently reliable. Scraps of paper don't stay in one place. Your digital device may be in the next county when you have an idea, so carry a journal.

Record Dates and Times

Someone once said, "Time is God's way of making sure everything doesn't happen at once." Effective testers use God's great invention to ensure they always know the order of past events. Put the date and time on everything you write down, everything you put in the computer, and everything the computer puts out. Record both the time of writing and the time of the event itself. It's an investment like insurance

it may seem a bit troublesome at the time, but when the accident occurs, you'll be glad you bought it. And even if there's never an accident, you've bought some peace of mind.

Standardizing Formats

When I write down the same thing in different ways, I always seem to have trouble later. Europeans write dates as day/month; Americans use month/day. Obviously, either method is workable, but used together they make a mess. After living in Europe, I came to realize that the European method was clearly more logical (for sorting order, at least), but that didn't mean I could adopt it when I returned to the US. It confuses Americans. Instead, I evolved a standard (for me) dating method—11 October 2017) which is far less likely to confuse either Europeans or Americans than 10/11, or is it 11/10?

My standard method also forces me to write the year. You'd be surprised how many bugs I've had that survived more than a year—or maybe you wouldn't be surprised at all.

Over long periods of time, my ideas about certain standards have changed, which some people use as an argument against any kind of standard. But one advantage of standard formats is that they allow easier transformation to a new standard, should that be necessary. And, if I've dated everything and recorded the change in standards in my journal, I can always figure out which standard applied to a particular item.

Filing

One of the areas in which I still could use some improvement is my filing. My journal often helps as an index to filed items, and dating every filed item was a great leap forward. But I still find myself too frequently in the situation where I know I've put away the solution to a problem but can't find it in my files.

Clearly, I'm in no position to tell other people how to file their myriad documents—test results, bug reports, articles, correction letters, and so forth—that might be needed when testing a system. But at least I speak with the experience of a common person rather than a filing genius (like a former secretary who filed the cans of food in her kitchen alphabetically).

Probably the most important principle I've learned is to think, when putting something away, about the question I'll be asking when I want to retrieve it. That question is my best guide as to where to file it. Recently, when I've been unable to reduce an important item to a single question, I've taken to filing a reference to the item in extra places. Or, if it's filed digitally, I add at least one keyword for each possible question.

Throwing Things Away

Filing multiple references tends to compound another problem of mine, based on the universal law: "Files Fill." I've noticed that good testers have a sense of when to rid themselves of things. I don't have this sense, and it has always cost me much wasted effort when searching for something.

When I used to work for IBM (which stood for "I've Been Moved"), every time I packed my files I would purge about 1/3 of my filed material. Now that I don't move as frequently, I tend to drown in my files.

Once in a while, though, I become seized with a cleaning fit and go through a drawer or two, or tidy up a few folders. Still, I'm still trying to learn something good about filing—or, rather, un-filing.

Learn Something New Before Sleeping

My father was an extraordinarily smart man. Somewhere in my early years, he told me that he would never go to bed at night unless he had learned something new that day. Somehow that made sense to me, perhaps because I wanted to be smart like my father. For whatever reason, I have followed this habit for my entire life. I was never one of those people who felt he could learn only in a classroom, so every night, before hitting the pillows, I review my day for learnings. My journal helps. If I cannot think of anything worth writing down, I go to my library and grab a book. Or, nowadays, I'll go online and browse until I find something of value.

Sometimes, it's something new. Others, it's a reminder of something I used to know, but have forgotten. Once in a great while, I try to retire without this learning, but I've discovered that this habit is so ingrained that I cannot fall asleep. So, I get up from bed and start the learning process over again.

Learning From Others

Which brings me to the final habit on my list, undoubtedly the one that has done more than all the others to make me a better tester.

I don't know how I acquired this habit—I certainly didn't have it when I was in school. Somewhere along the line, I developed the habit of listening to other people. And watching them.

Of course, I try to learn from my own mistakes, my ego makes it hard to learn from them. Besides there are so many other people making mistakes that I can learn even more from their blunders. I think that's how Dani learned how to train dogs—and dog owners.

Back To Index

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

James Bach says, "Read this book and get your head straight about testing. I consider Jerry (Weinberg) to be the greatest living tester."

Answers the questions that puzzle the most people:

Why do we have to bother testing?

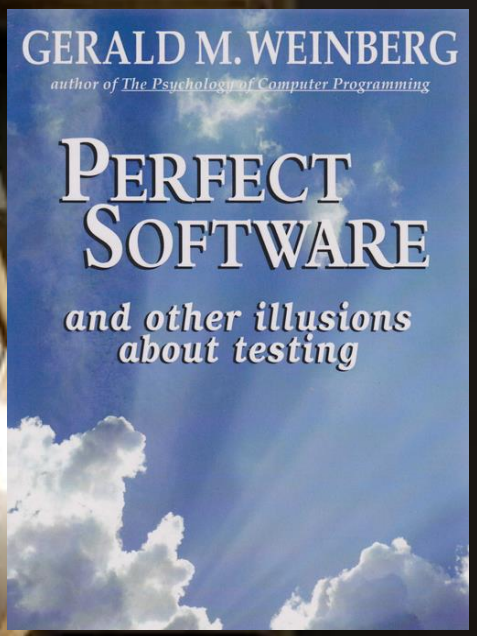
Why not just test everything?

What is it that makes testing so hard?

Why does testing take so long?

Is perfect software even possible?

Why can't we just accept a few bugs?



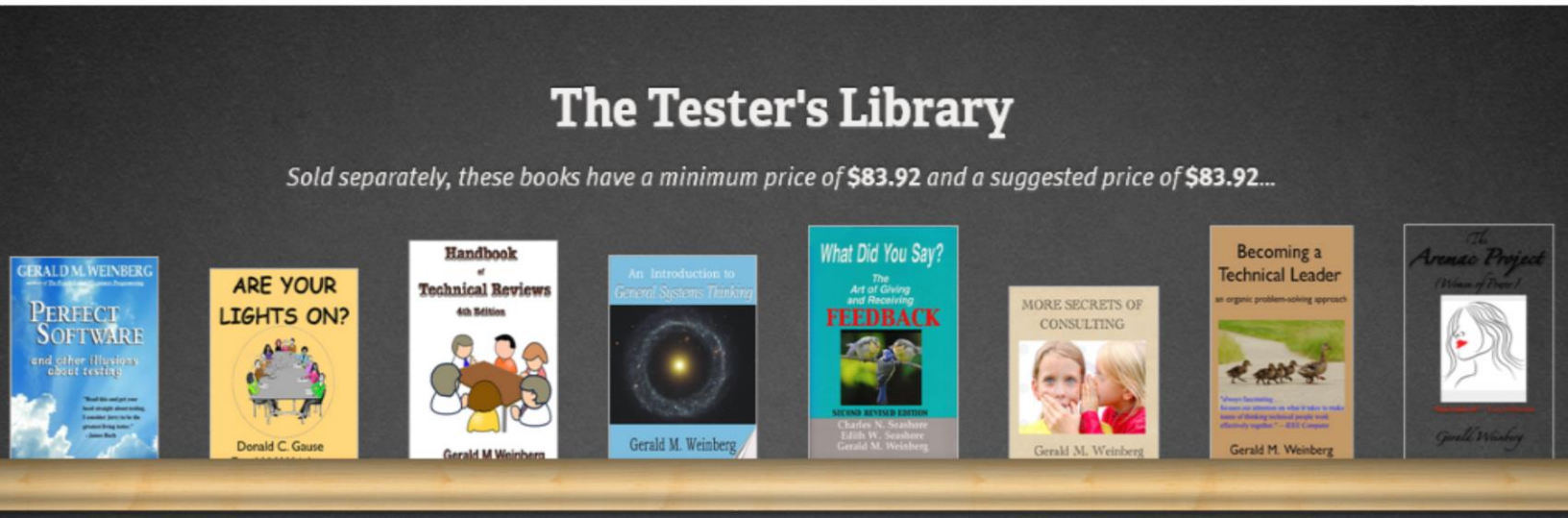
TTWT Rating: ★★★★★

A book that every tester should read.

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

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

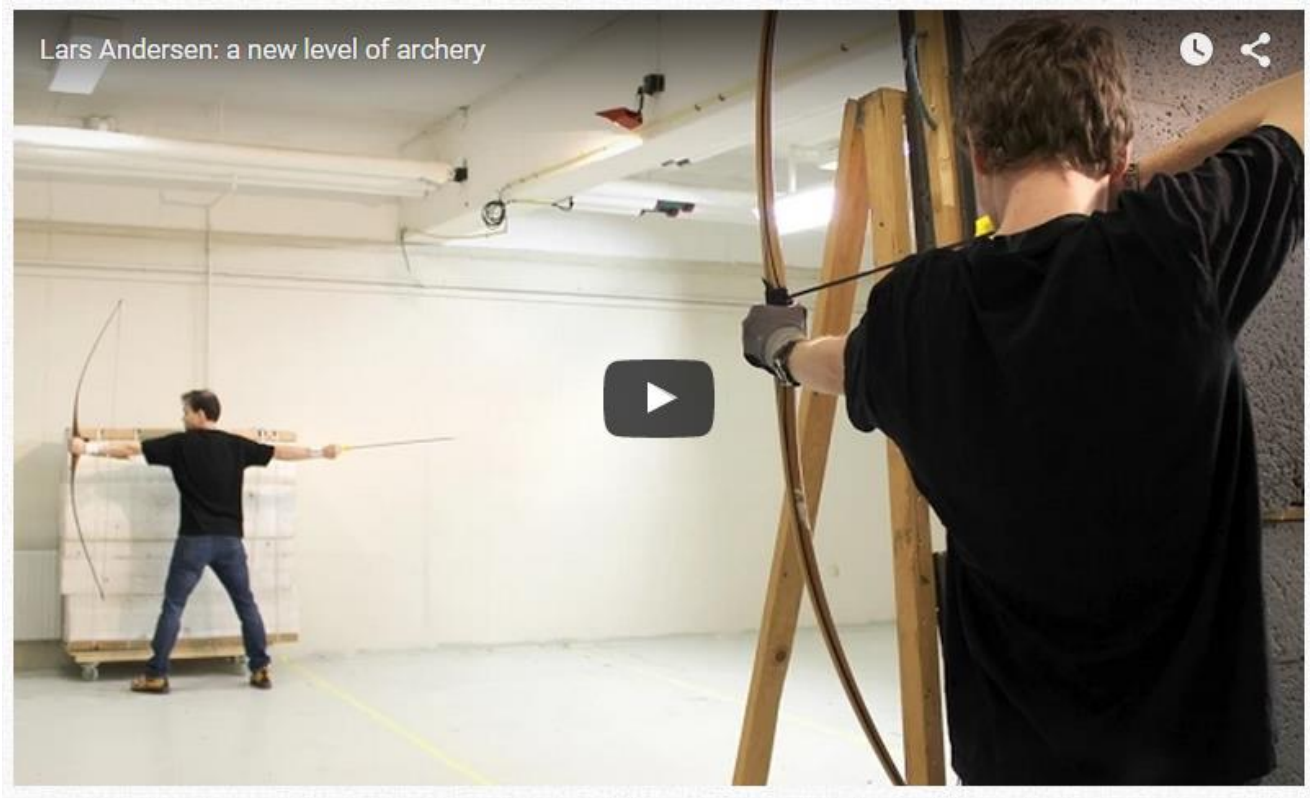
Know more about this bundle


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

A New Level of Testing?

- by Huib Schoots

Some months back, I saw this awesome video of Lars Andersen: a new level of archery. It was going viral on the web being watched over 11 million times within 48 hours. Now watch this movie carefully…



The first time I watched this movie, I was impressed. Having tried archery several times, I know how hard it is to do. Remember Legolas from the Lord of Rings movie? I thought that was "only" a movie and his shooting speed wasgreatly exaggerated. But it turns out Lars Andersen is faster than Legolas. My colleague Sander send me an email telling me about the movie I just watched saying this was an excellent example of craftsmanship, something we have been discussing earlier this week. So I watched the movie again…

Also read what Lars has to say in the comments on YouTube and make sure you read his press release.

This movie is exemplar for the importance of practice and skills! This movie explains archery in a way a context-driven tester would explain his testing…

0:06 These skills have been long since been forgotten. But master archer Lars Andersen is trying to reinvent was has been lost…

Skills are the backbone of everything being done well. So in testing skills are essential too. I'll come back to that later on. And the word reinvent triggers me as well. Every tester should reinvent his own testing. Only by going very deep, understand every single bit, practice and practice more, you will truly know how to be an excellent tester.

0:32 This is the best type of shooting and there is nothing beyond it in power or accuracy. Using this technique Larsen set several speed shooting records and he shoots more than twice as fast as his closest competitors…

Excellent testers are faster and better. Last week I heard professor Chris Verhoef speak about skills in IT and he mentioned that he has seen a factor 200 in productivity difference between excellent programmers and bad programmers (he called them "Timber Smurf" or "Knutselsmurf" in Dutch).

0:42 … being able to shoot fast is only one of the benefits of the method

Faster testing! Isn't that what we are after?

0:55 surprisingly the quiver turned out to be useless when it comes to moving fast. The back quiver was a Hollywood Myth…

The back quiver is a Hollywood myth. It looks cool and may look handy on first sight, since you can put a lot of arrows in it. Doesn't this sound like certificates and document-heavy test approaches? The certificates looks good on your resume and the artifacts look convenient to help you structure your testing… But turn out to be worthless when it comes to test fast.

1:03 Why? Because modern archers do not move. They stand still firing at a target board.

I see a parallel here with old school testing: testers had a lot of time to prepare in the waterfall projects. The basic assumption was that target wasn't moving, so it was like shooting at a target board. Although the target proved always to be moving, the testing methods are designed for target boards.

1:27 Placing the arrow left around the bow is not good while you are in motion. By placing your hand on the left side, your hand is on the wrong side of the string. So you need several movements before you can actually shoot…

Making a ton of documentation before starting to test is like several movements before you can actually test.

1:35 From studying old pictures of archers, Lars discovered that some historical archers held their arrow on the right side of the bow. This means that the arrow can be fired in one single motion. Both faster and better!

Research and study is what is lacking in testing for many. There is much we can learn from the past, but also from social science, measurement, designing experiments, etc.

1:56 If he wanted to learn to shoot like the master archers of old, he had to unlearn what he had learned…

Learning new stuff, learning how to use heuristics and train real skills, needs testers to unlearn APPLYING techniques.

2:07: When archery was simpler and more natural, exactly like throwing a ball. In essence making archery as simple as possible. It's harder how to learn to shoot this way, but it gives more options and ultimately it is also more fun.

It is hard to learn and it takes a lot of practice to learn to do stuff in the most efficient and effective way. Context-driven testing sounds difficult, but in essence it makes testing as simple as possible. That means it becomes harder to learn because it removes all the methodical stuff that slows us down. These instrumental approaches trying to put everything in a recipe so it can be applied by people who do not want to practice, make testing slow and ineffective.

2:21 A war archer must have total control over his bow in all situations! He must be able to handle his bow and arrows in a controlled way, under the most varied of circumstances.

Lesson 272 in the book Lessons Learned in Software Testing: "If you can get a black belt in only two weeks, avoid fights". You have to learn and practice a lot to have total control! That is what we mean by excellent testing: being able to do testing in a controlled way, under the most varied of circumstances. Doesn't that sound like Rapid Software Testing? RST is the skill of testing any software, any time, under any conditions, such that your work stands up to scrutiny. This is how RST differs from normal software testing.

2:36 … master archers can shoot the bow with both hands. And still hit the target. So he began practicing…

Being able to do the same thing in different ways is a big advantage. Also in testing we should learn to test in as many different ways as possible.

3:15 perhaps more importantly: modern slow archery has led people to believe that war archers only shot at long distances. However, Lars found that they can shoot at any distance. Even up close. This does require the ability to fire fast though.

Modern slow testing has led to believe that professional testers always need test cases. However, some testers found that they could work without heavyweight test documentation and test cases. Even on very complex or critical systems also in a regulated environment. This does require the ability to test fast though.

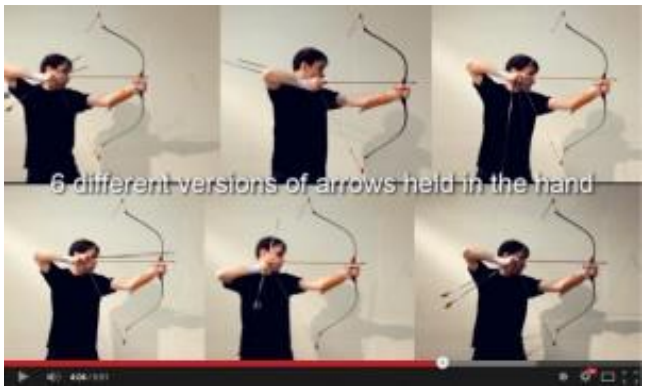
3:34 In the beginning archers probably drew arrows from quivers or belts. But since then they started holding the arrows in the bow hand. And later in the draw hand. Taking it to this third level. That of holding the arrows in the bow hand, requires immense practice and skill and only professional archers, hunters and so on would have had the

time for it. … and the only reason Lars is able to do it, is he has been years of practicing intensely.

Practice, practice, practice. And this really makes the difference. I hear people say that context-driven is not for everybody. We have to accept that some testing professional only want to work 9 to 5. This makes me mad!

I think professional excellence can and should be for everyone! And sure you need to put a lot of work in it! Compare it to football (or any other good thing you want to be in like solving crossword puzzles, drawing, chess or … archery). It takes a lot of practice to play football in the Premiership or the Champions League. I am convinced that anyone can be a professional football player. But it doesn't come easily. It demands a lot of effort in learning, drive (intrinsic motivation, passion), the right mindset and choosing the right mentors/teachers. Talent may be helps, and perhaps you need some talent to be the very best, like Lionel Messie … But dedication, learning and practice will take you a long way. We are professionals! So that subset of testers who do not want to practice and work hard, in football they will soon end up on the bench, won't get a new contract and soon disappear to the amateurs.

4:06 The hard part is not how to hold the arrows, but learning how to handle them properly. And draw and fire in one single motion not matter what methods is used.



Diversity has been key in context-driven testing for many years. As testers we need to learn how to properly use many different skills, approaches, techniques, heuristics…

4:12 It works in all positions and while in motion…

… so we can use then in all situations even when we are under great pressure, we have to deal with huge complexity, confusion, changes, new insights and half answers.

5:17 While speed is important, hitting the target is essential.

Fast testing is great, doing the right thing, like hitting the target is essential. Context-driven testers know how to analyze and model their context to determine what the problem is that needs to be solved. Knowing the context is essential to do the right things to discover the status of the product and any threats to its value effectively, so that ultimately our clients can make informed decisions about it. Context analysis and modelling are some of the essential skills for testers!

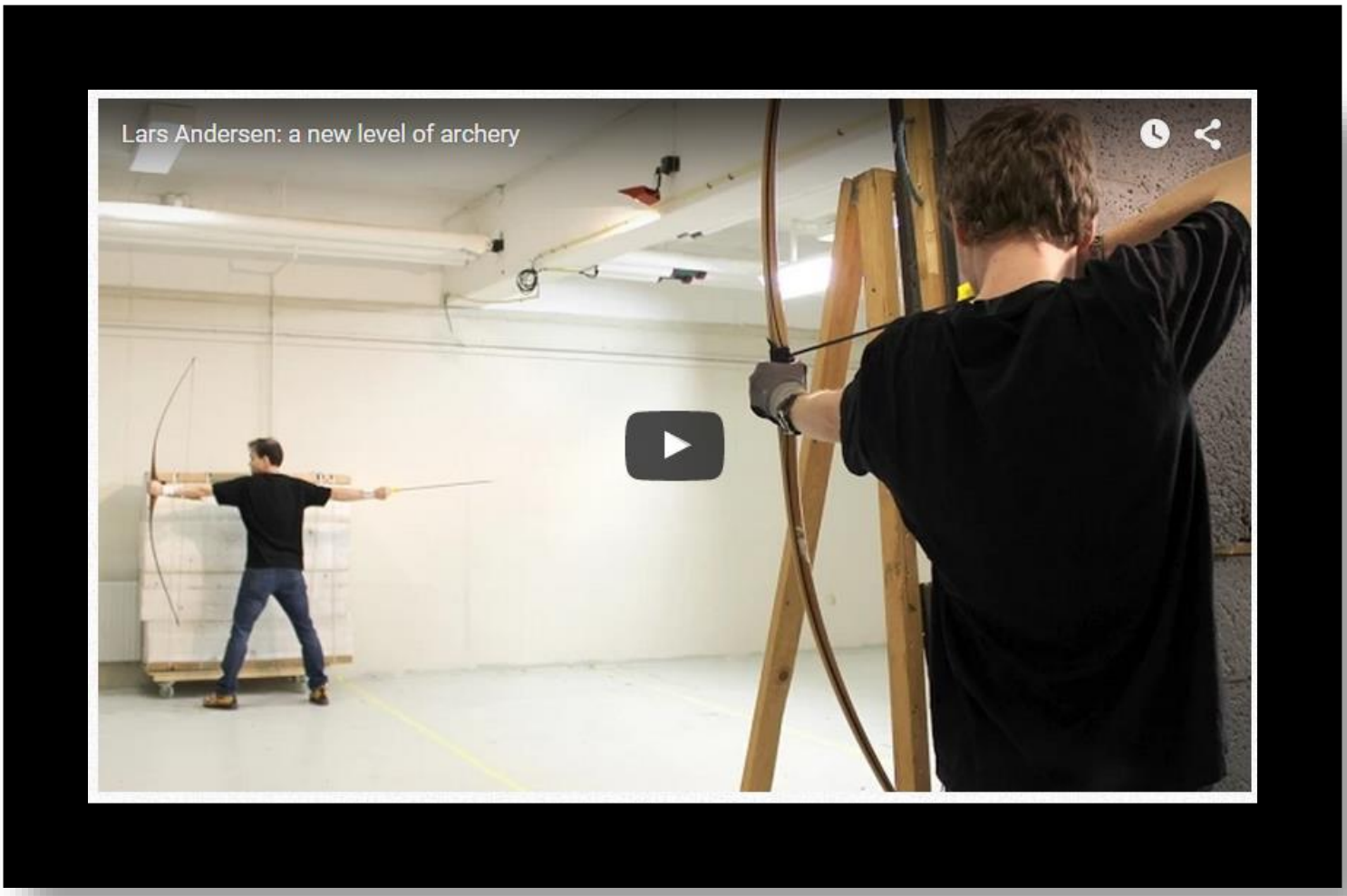
There are probably more parallels to testing. Please let me know if you see any more.

"Many people have accused me of being fake or have theories on how there's cheating involved. I've always found it fascinating how human it is, to want to disbelieve anything that goes against our world view – even when it's about something as relatively neutral as archery." (Lars Andersen)

Huib Schoots is a tester, consultant and people lover. He shares his passion for testing through consulting, coaching, training, and giving presentations on a variety of test subjects. With almost twenty years of experience in IT and software testing, Huib is experienced in different testing roles. Curious and passionate, he is an agile and context-driven tester who attempts to read everything ever published on software testing.

Huib is one of four instructors of Rapid Software Testing. A member of TestNet, AST and ISST, black-belt in the Miagi-Do School of software testing and co-author of a book about the future of software testing. Huib maintains a blog on magnifiant.com and tweets as @huibschoots. He works for Improve Quality Services, a provider of consultancy and training in the field of testing.

Back To Index



Love to Write?

Your ideas, your voice. Now it's your chance to be heard!

Send your articles to editor@teatimewithtesters.com


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

Leading Beyond The Scramble

A few years ago I was discussing an industry trend with a colleague. I wanted to move my career forward, to step up and grow beyond the testing niche I found myself stuck in. My friend listened patiently, and then at the end of the conversation, said "You know what Matt? Sometimes you have to step up and lead."

It was the right answer ... and I found it a bit empty. I wanted some advice on how to lead. "Just do it" left me wanting more.

When I sent my article on the scramble in for feedback, reading between the lines, I think my peer reviewers felt a bit like I had in that conversation. Part one doesn't tell people how to lead - and certainly not how to deal with resistance.

That's okay, or at least, I am okay with that. Part one was intended as an inspiration piece; to explain the difference between motion and reaction. That is all.

Still, people thought I could do better.

I think they were right.

Ironically, if I tell you want what to do, I would be creating a kind of follower. Worse, as I'm not there and you are, my advice would likely be bad. What I can do is tell you a story or two, of a time I worked with a team that was close to panic, and helped influence the direction to something more reasonable.

Let's get started.

Regression Testing in eCommerce Land

It was a major eCommerce site for a wholesaler. The team was executing well, getting new code to production every-other two week sprint. Our test/release process took a week, which wasn't bad for a team with 10% testers covering current and the previous version of all major browsers, plus android, iPhone, and iPad devices. To do it we modeled coverage and quality on a low tech testing dashboard we created with Google Spreadsheets.

Then I came in one Friday morning at it all fell apart.

Well, almost.

The plan was to start the morning with a one-hour test training session. I had a copy of Rob Sabourin's "Ten Sources of Test Ideas", and each tester was going to create one test idea from three of the categories, and we would discuss. This was the middle of one iteration, and we weren't going to ship to production for three more weeks.

Except that suddenly changed.

Management wanted to put the new features out now. As in tonight ... and to make the ship decision by noon. I could almost feel the condescension as one manager said "it sounds like we won't have time for your little tester training exercises thing.", but I did not let that happen.

Because I knew what would happen. The team would feel pressure and bounce around the spreadsheet, either leaving holes in coverage, or making all the coverage 1 or 2, which is essentially "I breathed on it and it did not fall over." In other words, we would scramble. At the end of the process we wouldn't be able to provide meaningful information to management, and we'd either whine about needing more time or management would make a ship decision without much value from testing. (Unless we stumbled on some bugs that were clearly show-stoppers, which, as I'll explain in a bit, would have been unlikely.)

The "cover everything a little" approach wasn't even smart. We knew the features that had gone in the last week; we knew the path-to-purchase that mattered - login to search to product page, add to cart and checkout. We could customize the coverage based on risk and how critical the features were, plus any other information that mattered, such as what the developers or management were worried about, or browser popularity.

I pushed back. I said we were going to use the meeting to come up with the kind of strategy to enable just this problem - to provide valuable information in one-tenth the time.

So yes, we had my "little training session." Each tester looked at browser popularity, new features, core features, and Robert Sabourin's list to come up with three charters to run sessions we estimated to be 20 minutes of length. We put those stickies on a wall and dot-voted them, then sorted by importance. Starting in the TODO column, each tester pulled one test idea at a time into DOING, then DONE, then worked on another, all morning. Management could see what we were working on, they could add new test ideas, they could move the priority around - and they could see what we left to be tested. At noon we got test and management together and took a poll by having the team "vote" (thumbs up, down, or middle) with if the ship decision should happen.

No one scrambled. No one felt pressured. The testers led that release, instead of following.

It was one of the finer moments of that year.

A Time I Opted Out ("Or What?")

The story above is true, and it was a good day. To balance things, I'd like to tell you about a not so good day.

It was a different project at a different company, but yet another scramble. The development team was writing reports against tables that did not exist; the test system wouldn't even boot up and run. It was bad. When I started on the project it was three months late, by which I do not mean the project was late, but that I started three months after it was supposed to have shipped to production.

We did not have a working test system.

Every three months the project management team would slip the schedule by three months. I would literally say "isn't it time for a three month slip?" and we would have one.

I have no idea what the technical staff were doing all day. They spent a great deal of effort trying to convince each other they were busy, even though in some cases the systems we were supposed to be working on were no operational. There was a lot of scrambling, of status reports and bluster. It didn't make any sense. I transferred out after about four months, and the project kept slipping.

The final go-live date was June 1st. Word came from on high that the team had to have things working by June 1st, or "else." I don't know what "or else" meant. I certainly didn't see a reason that we were going to go-live on that day.

I have to shamefully admit that I had some fun with the technical staff, going down to the team area and asking leading questions.

Matt: "So you're going to go-live on June 1st?"
MTS: "Yes, we will!"
Matt: "Or what?"
MTS: "We MUST go live on June 1st?"
Matt: "Or ... what?"
MST: "We HAVE to go live on Just 1st?"
Matt: "Right. But what if you don't?"

You guessed it. No go-live on June 1st.

The project slipped to July 1st (The "true, last, end-of-project, must happen go live date") ... and they didn't make July 1st either. Around the middle of July, the company cancelled the project, fired and threatened to sue the 3rd party vendor providing the engine we were trying to integrate with.

Like I said, this is not my finest moment. I probably should not have teased the people on the failing project as it was approaching the end-game. That wasn't very nice. I'm not super-happy about the outcome, either. Matt saved himself, but the rest of the team scrambled. I did not influence the outcome. Then again, looking back, I doubt I could have influenced the outcome. Getting out might have been the best I could do.

One thing I did learn from this is a heuristic to recognize when the scramble around you is a problem; the kind of scramble that will end in failure. These look, from the outside, like any other scramble: Everyone is going insane, running around, because they have to make the deadline. The difference is when you ask for real consequences to the business if the team misses the date. You know you're in trouble when no one can give you any. That means the scramble is likely artificial and low value. As Ed Yourdon explains in his book Death March - "Sometimes the tough artificial date is because the real value of the project is slow. Scheduling it realistically would make it not worth doing."

Don't be afraid to ask "What if we miss?" - And listen carefully to the answer.

No, It is not going to be the end of the world

Over the years I have heard technical (and business) people of all stripes tell me they "had" to follow, or cheat, or do something else they found distasteful "to feed my family", or "else I would be fired."

I'd like to suggest an alternative option besides mindless following.

First, breathe. Clear your mind. Take a walk.

Figure out the worst thing that could possibly happen. You get fired.

Ask yourself: Is that really going to happen? And if it did, so what? You'd get another job.

You might lose your house. So what? Can you get support from family, friends, your church, the government?

It is possible that the most that can happen is damage to your stuff, or your pride. And realistically, you won't lose that much stuff. You'll get another job. You likely won't even get fired. You might get a 0% raise one year. I certainly have. The earth will continue to revolve around the sun.

Once you realize that it won't be the end of the world, you realize that no, you do not have to follow, to give in to the scramble. You can, instead, lead. Propose reality the way you see it, act assertively that way, and other people will see things that way as well. Before you know it, people will look to you for answers.

It's been said before that the best way to be effective at your job is to act like you don't care about keeping it.

So forget about the scramble. Forget about the craziness. Take a breath, pause, then do what's best for the company.

A Final Thought

Sometimes you can't lead the way out of the scramble. People are excited, frantic, frenzied, going to stay that way, and there is nothing you can do about it. Keep your own wits about you, stay in control of what you can, and remember the name of the project. Once it's over, count the cost of project SquareCalc, or whatever it is, and remember the name.

Then, the next time the team is about to launch into a scramble, suggest that "Everyone wait a minute. Isn't this familiar? Isn't this just like project SquareCalc? Do we want another Squarecalc?"

Sometime the only way the team can learn, change and grow is through shared experience. You may have to let them experience it, and then remind them of it, all in a gentle, measured way.

Moving away from the scramble is hard to do, I know. Still, you wanted a real article about test leadership.

Whoop.

There it is.

Matt Heusser is the managing consultant at Excelon Development and editor in chief for Stickyminds.com. In addition to nearly two decades of professional work in software delivery, Matt is also the creator of the TestRetreat series of conferences, the creator of the lean software delivery courses and methods, lead organizer of the workshop on technical debt, the workshop on self-education in software testing (WHOSE), and the workshop on Teaching Test Design ("WhatDa"). A former member of the board of directors of the Association for Software Testing, Matt led the initial effort to create a grant program and headed the grant committee for its first two years. Space does not permit a full list of Matt's accomplishments, but you can learn more about Excelon at www.xndev.com , follow Matt on twitter @mheusser, or email him using Matt@xndev.com.

Meeting the Fundamental Challenges of Data Warehouse Testing: Part 1

- by Wayne Yaddow

Introduction

Decisions in today's organizations have become increasingly real time, and the systems that support business decisions must be high quality. People sometimes confuse testing data warehouses that produce business intelligence (BI) reports with backend or database testing or with testing the BI reports themselves. Data warehouse testing is much more complex and diverse than that. Nearly everything in BI applications involves data that is the most important component of intelligent decision making.

This article identifies three primary data warehouse testing challenges and offers approaches and guidelines to help you address them. The best practices and the test methodology presented here are based on practical experiences testing BI/DW applications.

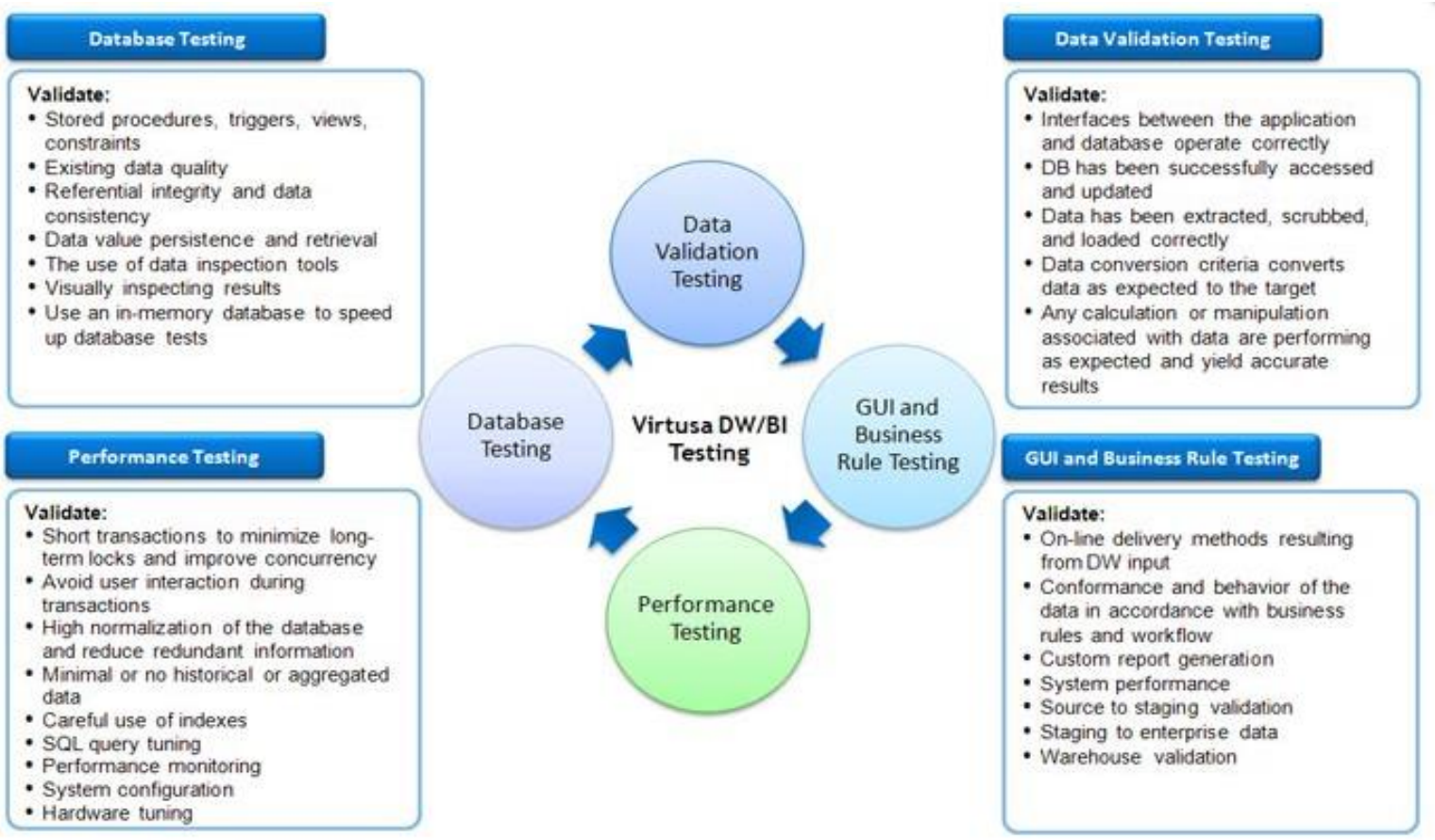


Figure 1: Testing the data warehouse – test focal points for quality assurance (graphic courtesy of Virtusa Corp.)

Not all data warehousing challenges are complex, but they are diverse. Unlike other software projects, data warehousing projects are not developed with a front-end application in mind. Instead, they focus on the back-end infrastructure that supports the front-end client reporting.

Knowing these challenges early, and using them as an agenda, provides a good technique for solving them. They fall into three major categories:

1. Identifying each necessary data warehouse test focus and associated project document needed for test planning

2. Confirming that all stages of the extract, transform, and load (ETL) process meet quality and functional requirements

3. Identifying qualified testing staff and skills

What does this mean for testing your data warehouse? Over time, a frequently changing and competitive market will raise new functional requirements for the data warehouse. Because enterprises must comply with new and changing compulsory legal and regulatory requirements, they need frequent releases and upgrades, so data and related applications must be tested several times throughout the year.

Before describing the challenges of testing data warehouses, we review some of the key characteristics that define most data warehouse projects:

- Diverse sources of data feed systems
- Complex environments containing multiple databases, file systems, application servers, programming tools, and database programming languages
- Wide variety of rules to process and load data
- Dynamic requirements for presenting or working with data
- Complex dataset types such as files, tables, queues, and streams
- Data may be structured, semi-structured, or unstructured
- High dependency on the interfacing systems and error/exception handling

Test planning for your DW/BI project should be designed to overcome the most significant challenges for data warehouse testers. We look at three such challenges in this article.

Challenge 1: Identifying tests and documentation for data warehouse test planning

Because data warehouse testing is different from most software testing, a best practice is to break the testing and validation process into several well-defined, high-level focal areas for data warehouse projects. Doing so allows targeted planning for each focus area, such as integration and data validation.

- **Data validation** includes reviewing the ETL mapping encoded in the ETL tool as well as reviewing samples of the data loaded into the test environment.

- **Integration testing** tasks include reviewing and accepting the logical data model captured with your data modeling tool (such as ERwin or your tool of choice), converting the models to actual physical database tables in the test environment, creating the proper indexes, and testing the ETL programs created by your ETL tool or procedures.

- **System testing** involves increasing the volume of the test data to be loaded, estimating and measuring load times, placing data into either a high-volume test area or in the user acceptance testing (UAT) and, later, production environments.

- **Regression testing** ensures that existing functionality remains intact each time a new release of ETL code and data is completed.

- **Performance and scalability tests** assure that data loads and queries perform within expected time frames and that the technical architecture is scalable.

- **Acceptance testing** includes verification of data model completeness to meet the reporting needs of the specific project, reviewing summary table designs, validation of data actually loaded in the production data warehouse environment, a review of the daily upload procedures, and finally application reports.

Few organizations discard the databases on which new or changed applications are based, so it is important to have reliable database models and data mappings when your data warehouse is first developed, then keep them current when changes occur. Consider developing the following documents most data warehouse testers need:

Source-to-target mapping. The backbone of a successful BI solution is an accurate and well-defined source-to-target mapping of each metric and the dimensions used. Source-to-target data mapping helps designers, developers, and testers understand where each data source is and how it transitioned to its final displayed form. Source-to-target mappings should identify the original source column names for each source table and file, any filter conditions or transformation rules used in the ETL processes, the destination column names in the data warehouse or data mart, and the definitions used in the repository (RPD file) for the metric or dimension. This helps you derive a testing strategy focused more on the customized elements of the solution.

Data models. Data warehouse models are crucial to the success of your data warehouse. If they are incorrect or non-existent, your warehouse effort will likely lose credibility. All project leaders should take the necessary time to develop the data warehouse data model. For most data warehouses, a multi-month building effort with a highly experienced data warehouse modeler may be needed after the detailed business requirements are defined. Again, only a very experienced data warehouse modeler should build the model. It may be the most important skill on your data warehouse team.

The data architecture and model is the blueprint of any data warehouse and understanding it helps you grasp the bigger picture of a data warehouse. The model helps stakeholders understand the key relationships between the major and critical data sources.

We stress the importance of getting your data model right because fixing it might require a great deal of effort, in addition to stalling your project. We've seen projects with models so corrupted that it almost makes sense to start from scratch.

Change management. Several factors contribute to information quality problems. Changes in source systems often require code changes in the ETL process. For example, in a particular financial institution, the ETL process corresponding to the credit risk data warehouse has approximately 25 releases each quarter. Even with appropriate quality assurance methods, there is always room for error. The following types of potential defects can occur when ETL processes change:

- Extraction logic that excludes certain types of data that were not tested.
- Transformation logic may aggregate two different types of data (e.g., car loan and boat loan) into a single category (e.g., car loan). In some cases, transformation logic may exclude certain types of data resulting in incomplete records in the data warehouse.
- Current processes may fail due to system errors or transformation errors resulting in incomplete data loading. System errors may arise when source systems or extracts are not available or when source data has the incorrect format. Transformation errors may also result from incorrect formats.
- Outdated, incomplete, or incorrect reference and lookup data leads to errors in the data warehouse. For example, errors in the sales commission rate table may result in erroneous commission calculations.
- Data quality issues including source system data that is incomplete or inconsistent. For example, a customer record in the source system may be missing a ZIP code. Similarly, a source system related to sales may use an abbreviation of the product names in its database. Incompleteness and inconsistency in source system data will lead to quality issues in the data warehouse.

Challenge 2: Identifying the precise focus of ETL testing

Well-planned extraction, transformation, and load testing should be a high priority and appropriately focused. Reconciliation of warehouse data with source system data (data that feeds the data warehouse) is critical to ensure business users have confidence that all reported information is accurate according to business requirements and data mapping rules.

Why is ETL testing so complex? Most of the action occurs behind the scenes; output displayed in a report or as a message to an interfacing system is just the tip of the iceberg. There is always more underlying data behind such reports and messages. The combinations can be virtually endless depending on the type of application and business/functional logic. Furthermore, enterprises are dependent on various business rules as well as different types of data (such as transactional, master, and reference data) that must be considered.

The environments to be tested are complex and heterogeneous. Multiple programming languages, databases, data sources, data targets, and reporting environments are often all integral parts of the solution. Information for test planning can come from functional specifications, requirements, use cases, user stories, legacy applications, and test models—but are they complete and accurate? What about

conflicts in input specifications? Many data types must be managed, including files, tables, queues, streams, views, and structured and unstructured datasets.

How do you represent test data? How do the testers prepare test data? How do they place it where it belongs? How much time will this take, and how much time *should* it take?

You can perform testing on many different levels, and you should define them as part of your ETL testing strategy. Examples include the following:

Constraint testing. The objective of constraint testing is to validate unique constraints, primary keys, foreign keys, indexes, and relationships. Test scripts should include these validation points. ETL test cases can be developed to validate constraints during the loading of the warehouse. If you decide to add constraint validation to the ETL process, the ETL code must validate all business rules and relational data requirements. If you automate testing in any way, ensure that the setup is done correctly and maintained throughout the ever-changing requirements process. An alternative to automation is to use manual queries. For example, you can create SQL queries to cover all test scenarios and execute these tests manually.

Source-to-target count comparisons. The objective of "count" test scripts is to determine if the record counts in corresponding source data matches the expected record counts in the data warehouse target. Counts should not always match; for example, duplicates may be dropped or data may be cleansed to remove unwanted records or field data. Some ETL processes are capable of capturing record count information (such as records read, records written, or records in error). When the ETL process can capture that level of detail and create a list of the counts, allow it to do so. It is always a good practice to use SQL queries to double-check the source-to-target counts.

Source-to-target data validation. No ETL process is smart enough to perform source-to-target field-to-field validation. This piece of the testing cycle is the most labor-intensive and requires the most thorough analysis of the data. There are a variety of tests you can perform during source-to-target validation. For example, verify that:

- Primary and foreign keys were correctly generated
- Not-null fields were populated properly
- There was no improper data truncation in each target field
- Target table data types and formats are as specified

Transformations of source data and application of business rules. Tests to verify all possible outcomes of the transformation rules, default values, and straight moves as specified in the business requirements and technical specification document.

Users who access information from the data warehouse must be assured that the data has been properly collected and integrated from various sources, after which it has been transformed in order to remove inconsistencies, then stored in formats according to business rules. Examples of transformation testing include the following:

- Table look-ups. When a code is found in a source field, does the system access the proper table and return the correct data to populate the target table?

- Arithmetic calculations. Were all arithmetic calculations and aggregations performed correctly? When the numeric value of a field is recalculated, the recalculation should be tested. When a field is

calculated and automatically updated by the system, the calculation must be confirmed. As a special mention, you must ensure that when you apply business rules, no data field exceeds its boundaries (value limits).

Batch sequence dependency testing. Data warehouse ETLs are essentially a sequence of processes that execute in a defined sequence. Dependencies often exist among various processes. Therefore, it is critical to maintain the integrity of your data. Executing the sequences in the wrong order might result in inaccurate data in the warehouse. The testing process must include multiple iterations of the end-to-end execution of the entire batch sequence. Data must be continually verified and checked for integrity during this testing.

Job restart testing. In a production environment, ETL jobs and processes fail for a variety of reasons (for example, database-related or connectivity failures). Jobs can fail when only partially executed. A good design allows for the ability to restart any job from its failure point. Although this is more of a design suggestion, every ETL job should be developed and tested for restart capability.

Error handling. During process validation, your testing team will identify additional data cleansing needs and identify consistent error patterns that might be averted by modifying the ETL code. It is the responsibility of your validation team to identify any and all suspect records. Once a record has been both data and process validated and the script has passed, the ETL process is functioning correctly.

Views. You should test views created with table values to ensure the attributes specified in the views are correct and the data loaded in the target table matches what is displayed in the views.

Sampling. These tests involve selecting a representative portion of the data to be loaded into target tables. Predicted results, based on mapping documents or related specifications, will be matched to the actual results obtained from the data loaded. Comparison will be verified to ensure that the predictions match the data loaded into the target table.

Duplicate testing. You must test for duplicates at each stage of the ETL process and in the final target table. This testing involves checks for duplicate rows and checks for multiple rows with the same primary key, neither of which can be allowed.

Performance. This is the most important aspect after data validation. Performance testing should check that the ETL process completes within the load window specified in the business requirements.

Volume. Verify that the system can process the maximum expected data volume for a given cycle in the time expected. This testing is sometimes overlooked or conducted late in the process; it should not be delayed.

Connectivity. As its name suggests, this involves testing the upstream and downstream interfaces and the intra-data-warehouse connectivity. We suggest that the testing examines the exact transactions between these interfaces. For example, if the design approach is to extract the files from a source system, test extracting a file from the system, not just the connectivity.

Negative testing. Check on whether the application fails and where it should fail with invalid inputs and out-of-boundary scenarios**.**

Operational readiness testing (ORT). This is the final phase of testing; it focuses on verifying the deployment of software and the operational readiness of the application. During this phase you will test:

- The solution deployment
- The overall technical deployment "checklist" and timeframes
- The security of the system, including user authentication and authorization and user access levels

Evolving needs of your business and changes in source systems will drive continuous change in the data warehouse schema and the data being loaded. Hence, development and testing processes must be clearly defined, followed by impact analysis and strong alignment between development, operations, and the business.

Challenge 3: Choosing qualified testers for data warehouse QA efforts

The impulse to cut costs is often strong, especially in the final delivery phase. A common mistake is to delegate testing responsibilities to resources with limited business and data testing experience.

Our best-practice recommendations to help you choose the best testers follow.

Identify crucial tester skills. The data warehouse testing lead and other hands-on testers are expected to demonstrate extensive experience in their ability to design, plan, and execute database and data warehouse testing strategies and tactics to ensure data warehouse quality throughout all stages of the ETL life cycle.

Recent years have seen a trend toward business analysts, ETL developers, and even business users planning and conducting data warehouse testing. This may be risky. Among the required skills for data warehouse testers are the following:

- A firm understanding of data warehouse and database concepts
- The ability to develop strategies, test plans, and test cases specific to data warehousing and the enterprise's business
- Advanced skill with SQL queries and stored procedures
- In-depth understanding of the organization's ETL tool (for example, Informatica, SSIS, DataStage, etc.)
- An understanding of project data and metadata (data sources, data tables, data dictionary, business terminology)
- Experience with data profiling with associated methods and tools
- The ability to create effective ETL test cases and scenarios based on ETL database loading technology and business requirements
- Understanding of data models, data mapping documents, ETL design, and ETL coding
- The ability to communicate effectively with data warehouse designers and developers
- Experience with multiple DB systems, such as Oracle, SQL Server, Sybase, or DB2
- Troubleshooting of the ETL (e.g., Informatica/DataStage) sessions and workflows
- Skills for deployment of DB code to databases
- Unix/Linux scripting, Autosys, Anthill, etc.
- Use of Microsoft Excel and Access for data analysis
- Implementation of automated testing for ETL processes
- Defect management and tools

Conclusions

We have highlighted several approaches and solutions for key data warehouse testing challenges following a concept-centered approach that expands on successful methods from multiple respected resources. The testing challenges and proposed solutions described here combine an understanding of the business rules applied to the data with the ability to develop and use QA procedures that check the accuracy of entire data domains—i.e., both source and data warehouse targets.

Suggested levels of testing rigor frequently require additional effort and skilled resources. However, employing these methods, DW/BI teams can be assured from day one of their data warehouse implementation and data quality. This will build confidence in the end-user community and will ultimately lead to more effective DW/BI implementations.

Testing the data warehouse and BI applications requires good testing skills as well as active participation in requirements gathering and design phases. Additionally, in-depth knowledge of BI/DW concepts and technology is crucial so that one may comprehend the end-user requirements and therefore contribute to a reliable, efficient, and scalable design. We have presented the numerous flavors of testing involved while assuring quality of BI/DW applications. Our emphasis relies on the early adoption of a standardized testing approach with the customization required for your specific projects to ensure a high-quality product with minimal rework.

Wayne Yaddow is a senior data warehouse, ETL, and BI report QA analyst working as a consultant in the NYC financial industry. He has spent 20 years helping organizations implement data quality and data integration strategies.

Wayne can be reached at wyaddow@gmail.com

Back To Index

Notes on Test Automation: Test Encapsulation

Enabling Self-Aware Automated Test Cases: Part 2

by Rahul Verma

4. Did We Miss Discussing Something?

In the discussion of this complex and long list of requirements, we missed the most important thing. What's the final purpose of a TAF? With all the "noise-features", which help in scheduling, executing and reporting with an icing of user-friendliness, at the core of it, *a TAF is about running one or more tests*. Where in our discussion did we talk about what the test should look like? How often does this aspect get discussed when we talk about TAFs?

TAFs are like onions. Their tastiest part is their core covered with lots of wrappers/layers. For a TAF the tastiest (the most important) part is the code written by a tester which is going to execute the actual test on the system. In our TAF discussions, we are often so caught up in the wrappers and layers, that by the time we reach the heart of the problem, we have made it powerless and small, putting most of the things in the hands of the surrounding wrappers/layers.

If the "test" part of a TAF is not designed well, all other features will be useless. All user-defined configurations, default framework configurations, platform conditions, tool decisions, pass/fail decisions etc. have to be translated into how the test gets executed and how it should report the results back.

Because of this faulty approach, most of the information, which should enable the test to take decisions at runtime, is made available outside of the test, so that some outside component and not the test itself makes these decisions. As mentioned in the introduction, this can become highly problematic when you try to add more and more features to the framework.

5. Encapsulation

As per Grady Booch, encapsulation is – "The process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation."

If that sounds complex, let's settle for this:

When we encapsulate X, we provide inside the X:

- All data that defines the state of X as well as the data it requires for doing its job.
- All methods that X needs to use and manipulate the data available. Then we expose a subset of these methods as the external/contractual interface which the calling components would use to exercise its behavior.

6. Test Encapsulation

With test encapsulation you provide inside the test all the data it needs for test execution and all the methods related to setting-up, running, reporting and clean-up.

What does a test need to KNOW?

Let's personify test and see what it asks:

Meta data

Every test is unique in that it would have at least one thing that's different from the other tests in that framework. (If it is not, please check your review process!). It means, every test should be identifiable as a separate entity.

A test could ask the following questions in order to find out what it is:

- What is my name?
- What is my identification number? (a single test could have multiple IDs in different contexts)
- What is my purpose?
- Who is my author?
- What is my birth (creation) date? On which date, did I take my present shape? (Modification date)
- What is my version number?
- Who are my parents and grandparents? (parent group/sub-groups)

- What type of test am I?
- At which level do I work? (unit/component/system)
- What kind of software attribute do I test? (functional/performance/security)
- When should I get executed? (BVT/acceptance/main)
- Am I based on custom extensions? (e.g. CPPUnitLite based test)

Logging options

Logging of test results, progress of execution and dumping key information at precise locations is one of the key aspects of test automation.

The following are questions that a test might ask:

- Do I use the centralized logging mechanism or my own?
- Am I running in debug mode, thereby increasing what I log and changing where I log?
- Should I log performance statistics and local resource utilization statistics?

For runtime

Prior to test execution, a test must know some important things about itself to take decisions at runtime. Compare this to a game show, in which the person asking the questions would request that only those that satisfy a particular condition would be eligible for that round of the game. So, when he shouts "All those in black shirts", the ones in black shirts would rush towards the podium. But before getting up, the person must know that he is wearing a black shirt!

On the same lines, at runtime, the test would come to know about configuration settings chosen and the test environment around itself. At that time, to make important runtime decisions, a test would ask:

- How important am I? What is my priority?
- What is my version? Yes! I will utilize this when version based regression is on.
- Did I uncover some bugs so far? If so, what are their IDs?
- For which API versions of the product can I and can't I run?
- On which platforms can I or can't I run?
- Do I need stubs to run? Can I run at all if stub mode is on?

(There could be a requirement that you want to use stubs instead of actual components, which is usually the case with unit and component testing. There could be tests, which are meant to be run

only with stubs, others which can run with and without stubs and still others which would not run with stubs.)

From runtime configuration

There are various runtime configurations, which can be configured by the user or are set as defaults by the framework. As discussed in the previous section, these should get passed on by the framework to the test so that it can match them against its properties to take runtime decisions. One important example of such a decision is whether the test should get executed or not.

Let's see what kind of questions the test might ask:

- What is the API version under test?
- What is the platform on which I am being asked to run?
- What are the regression options? Are you looking for priority based regression, bug regression, version regression, author based regression or any other form?
- I have been asked to always log performance data. Do you want to override this?
- Do you want me to run in debug mode?
- What is the base directory reference path from which I am getting executed?
- What is the current mode of execution? Are you using stubs?

Execution properties

Once the test takes the decision that it is meant to execute (i.e. the criteria received from the runtime framework configuration settings match its own filter properties), there are execution related questions, which the test would need to ask:

- Where is the build under test located?
- Where are the tools that I need?
- How many threads/processes should I launch?
- The previous test has just completed. Should I delay execution by going into sleep mode?
- What configuration should I use for the tools that I plan to use?
- Where are my input files?
- To whom should I hand over the test results? Can I talk to the database directly or is there a middleman?

During and post execution

There are properties of the test which are set during and after execution of the test. These make up the basis of the following questions:

- How much time did I take to execute?
- Where am I in terms of execution? What is the current step I am executing?
- Where should I maintain my state information if I am asking for a reboot now?
- Did I ask for a system reboot/hibernate at a previous step?
- How many assertions have I made? How many of them passed or failed?
- Did any of the assertions relate to errors in execution?
- What is my final say about the test I executed? Did it pass or fail?
- Should I send notification to the administrator that I potentially succeeded in finding a bug?
- Should I send notification that I couldn't execute due to error in execution?

What does a test need to DO?

In section 5.1, we talked about all that a test needs to KNOW at runtime. Let's now look at all it needs to DO at runtime:

Prepare

A test needs to prepare itself. This means that at runtime the test needs to load all properties that were set for it statically by the writer of the test. It should also get hold of the runtime configuration and test environment settings passed to it by the test framework.

Set-up

It needs to do the initial set-up for the test that needs to be executed. For example, if the test needs something to be installed so that it can run, the set-up step should, as a pre-cursor to the test, complete the installation.

Run

This step in the test would include the code that executes the actual test. This could comprise of multiple assertions (sub-tests).

Report

The test would report the results of the test as per the configuration, e.g. to a chosen file or database, or publish the results to an object, or return them as part of the function call.

Clean-up

The test would clean up anything it created specifically for itself, so that the system is returned to its previous state ready for the next test.

7. How to Achieve Test Encapsulation?

What we discussed in Section 6 might seem like a lot to be put in a test, might it not? Relax! The fact that a test needs to know and do a lot does not mean that the solution has to be complex. You can still hide most of the complexity from the end user, thereby making the test scripts simple, yet powerful. The following are some quick tips:

- If you take a careful note of what a test should know, you will observe that most of it is optional and would be used only for certain test cases.
- You can set the meta data and other properties in a container to which the tests are registered. This means that this would not appear for every test case written, but still can be overridden at the test level, if needed. For example, a test group can have the author name as the XYZ, to which 30 test cases are subscribed and not even one of them needs to have the author property set by the script writer. Later, if another tester adds a test case, the author property can be overridden for the 31st test case.
- From all that the test should do, you'll find that, apart from prepare() and run(), all other commands are optional and can be set for the container if the setup() and tear-down() are common. In fact, grouping the tests like this into containers would make it easy to manage.

Similarly, the runtime checking for whether a test is "runnable" or not, is hidden from the end user, and is placed within the core library.

8. Benefits of Test Encapsulation

As a first benefit to reap once you have taken a decision to proceed with test encapsulation, you have already crossed the first hurdle by making the tests the most complex and powerful part of your framework.

In the introduction section, I mentioned the problem of running a test on multiple platforms and difficulties with the approach of folder wise grouping. Let's try to solve that with test encapsulation.

- Imagine that you created a test that knows on which platforms it is meant to run. Let's call it the MEANT-FOR-PLATFORMS property of the test, which is list or array of platform strings. For this example, let's say it is: ["WINXP_X86","WIN7_AMD64"], wherein the values signify that this test is meant to be executed on 32-bit Windows XP and 64-bit Windows 7 platforms.
- All tests reside together in a single folder structure; no redundant copies are created for the above-mentioned platforms.
- In the test cycle, when the TAF is running tests on a given test runner machine, it already knows the platform of that machine. While it is looping over the tests, it would pass this information to the test, for example, by setting the test's RUNNER_PLATFORM property.
- Now when the TAF asks the test to execute, the test can search for RUNNER_PLATFORM in its MEANT-FOR-PLATFORMS array, and only if it is found in the list, will the test agree to execute. Accordingly, the test would execute or the TAF would skip to the next test in queue.

In a similar way, you can add support for any runtime decisions. The approach is pretty much the same:

- Encapsulate the properties that form the basis of runtime decision in the test
- Make the base value available in advance
- Make the value against which comparison should be done at runtime available at runtime!

The second benefit that you get with test encapsulation is that your framework does not dictate to a test which product it is testing, which tools it should use with what parameters/configuration, how it should analyse the results or how it should report back etc. You can still provide these features which the test writer is free to use or ignore at will. This puts a lot of flexibility into the hands of the test writer, and you are free to enforce rules if the test writer chooses to use a given framework feature. An example is that a test writer might choose to write the report in a format and place of choice, but if he wants to use the web reporting platform provided by your framework (if it's there), then he has to provide the results in a pre-defined format.

The third benefit is that whatever your test does can be kept independent of other tests. It can set up and clean up any changes to the system. Also, you can put exception handling in the caller covering all individual test methods being called. This will enable the framework to continue execution, even if there's an error in a given test which has been called. These exceptions can be reported in the final test report generated by the framework and would help in providing accurate results as well as point testers, to the areas which need investigation.

9. Execution Time Overhead of Test Encapsulation

As you have probably observed by now, to skip a test with test encapsulation, involves a runtime decision. This means any skipping of the test would have to be done at runtime. Before this happens, the TAF must have already called the test constructor.

Being from the performance testing background, I can relate to any concerns relating to overhead, but this is not as much as it sounds. It doesn't take minutes or even seconds to create a test object. The time taken is a tiny fraction of a second. I designed a framework with this approach, and my requirement was to measure the time taken by the test to execute. For some tests, the values were not captured even with a precision of 6 decimal places when the unit was seconds.

Having said that, you can reduce this overhead still further:

- As discussed in section 6, you can group tests in containers. You could put the filter at the container level rather than at individual test level. For example, you can set the MEANT-FOR-PLATFORMS property for ABCTestGroup to which all tests of ABC category subscribe.
- When using this approach you have unlimited test filters. Look into the order in which filtering is done. Place the most commonly used filters first. For example, if the probability of an author based regression or a creation date based regression is much lower than bug regression in your context, filtering based on empty bug lists should come first.
- Test encapsulation has given you the capability to take runtime decisions, but it does not stop you from making careful exceptions. If you feel that most of your tests can be easily split up into platform-wise categories and there's minimal overlap, you could still group them into folders. Then resort to scheduling only specific tests for specific platforms. If over a period of time you observe that the scope of overlapping tests is increasing considerably, all you have to do is set the optimum platform property for tests and start relying on runtime checks.

So, the overall benefit of test encapsulation is that once you have carefully built it into your framework and created the underlying skeletal support, you can still go ahead with your traditional approach to test automation.

Rahul Verma is a consulting software tester, author, speaker, coach and a serial entrepreneur from Bangalore, India. He is the founder of Test Mile and Talent Reboot.

He is known for his practical and unified view of the software testing subject. He has been awarded multiple Testing Thought Leadership awards for his contributions to software testing community.

You can visit his website www.rahulverma.xyz to know more about his work and get in touch.

Back To Index


From Special Desk

TEA-TIME WITH SMARTBEAR

About this column…

SmartBear Software not only provides testing tools to help development and testing teams accomplish their software quality goals, it is also a hub of information and news for the software testing industry. From workflow methodologies to discussions on industry practices and tech conference coverage, SmartBear has become a source for testers seeking quick access to a wide variety of content.

SmartBear's goal in creating this column in **Tea-Time with Testers** is to empower software testers around the globe by helping them become more informed about the current state of the software testing industry.

Being Ready for IoT Success

- by **Paul Bruce**

It's one thing to get some devices to work together using APIs in a development shop; it's an entirely different thing to ship usable, innovative technology to consumers in a $14.4 trillion dollar market.

The explosion of manufacturer and developer interest in IoT-related services definitely means that there is going to be a lot of code and hardware that needs to be properly tested and monitored. Enterprises and startups alike are scrambling to pick up a piece of the action, but big names like IBM, Cisco, and Intel have been preparing for the IoT onslaught for years.

We must learn from our failures…

Many smaller innovators are simply trying to make sure that consumers aren't disappointed with IoT promises like they were with 3DTV, Google Glass, and drones. These kinds of cool technologies were simply not thought through enough, running into fundamental problems with nausea, privacy, and personal safety.

The IoT and API industries are more than just technological bedfellows, they share common philosophical underpinnings:* Great IoT devices should be secure, much like APIs should consider security throughout their architecture

- Great APIs should be composable with other APIs, just like multi-use IoT devices should be easy to integrate together
- APIs and IoT devices should maintain what industry leaders call "bounded context", to keep complexity manageable
- Great consumer services integrate IoT devices and external APIs to compound their usefulness to consumers

We must act on what we learn…

Even well thought out solutions that are designed and delivered responsibly can easily fail to stand up to demand. IoT devices and APIs need to be tested for performance characteristics, to understand how the system as a whole behaves under adverse or abnormal conditions.

Load testing your APIs is a solved problem, it's just a matter of making sure it is a priority and properly scheduled well before release. Load testing exposes problems in latency, concurrency, and reliability of both the code and the infrastructure, so by definition it is something that you want to put your DevOps hat on for. It is not solely a "QA problem".

Likewise, API virtualization is critical for teams to function autonomously and to see how connected IoT devices function when service behavior becomes less than stellar. We know what it looks like to the service when there are too many devices connected to it, but what does an unreliable API look like to devices and apps? API testing teams now need control over the situations they can simulate to make sure the overall system as a whole is truly ready for production rollout.

When we act on what we learn, things get much better…

IoT is poised to put more new devices and services in play than cloud, hardware virtualization, or mobile initiatives ever did combined. There's plenty of money to be made, but only by those who are quick to innovate and disciplined enough to test all aspects of their solutions stand to benefit from the IoT wave.

Please for the love of whatever, let's build a connected world that is safe, reliable, resilient to change, and one that ultimately makes us happier people. Let's be creative and successful, but careful and considerate at the same time.

Back To Index

Sharing is caring! Don't be selfish ☺

Share this issue with your friends and colleagues!

CONTINUED FROM PAGE 9...

Thanks, Cem. Now a question for individual tester's career; how should a tester improve her prospects in software testing?

I hear this question most often from freshers, people who have recently gotten a job in testing or who are trying to get one. My impression of the typical person who asks this question is of someone who is intelligent, ambitious, willing to work hard, maybe university educated but without many skills that are well enough developed to be of practical value in the workplace. This person is at the start of a journey of self-improvement that will require many years of work.

I think the specific path will be unique for each person, but every successful path will involve both, deepening your knowledge and skills and diversifying knowledge, skills, and the ways you can apply them.

What do you see as the basic knowledge that freshers should have?

A smart fresher should be able to do basic bug-hunting in a few weeks. I think there are four central tasks:

1. Learn a variety of tricks for exposing the relatively easy to find bugs. I think Elisabeth Hendrickson's book, Explore It! provides an excellent starting point. For testers who have a stronger knowledge of programming, I also highly recommend James Whittaker's How to Break… series of books. Adding depth over time will involve learning more of the ways that programs fail and common ways to look for each type of failure. This is the essence of risk-based testing.

2. Learn the vocabulary of software testing and the basics of the main testing techniques. You'll quickly discover that the vocabulary is inconsistent. Different people mean different, sometimes contradictory, things when they use the same words. Similarly, the basic descriptions of the techniques are sketchy and inconsistent. Adding depth over time with specific techniques will come from practical experience, including coaching by other testers, digging through books and searching the web for detailed examples (try them out yourself before relying on them—you will find specific, detailed suggestions that don't actually work very well)

3. Learn good data handling skills and other good work habits. Learn how to organize your time, so you know what tasks you have to do and you keep track of what you have done and what is left. Learn how to track how you are spending your time and how long different tasks take you. Pay attention to your mistakes. What were they? Why did you make them? How can you improve? Learn how to describe your test results and your work status (your progress so far and the work you have left to do). Find a way to communicate professionally, giving accurate information without whining, pleading, blaming or making excuses.

4. Learn your subject area. For example, if you are testing software that relies heavily on statistics (all of data mining relies heavily on statistics), you probably want to learn more about math and more about the management of complex collections of data. As you understand more about the subject area, you will be better able to imagine what can go wrong with it.

I think most commercial testing courses teach at the surface level, even some of the "advanced" ones. That is, I think they teach basic bug-hunting methods, testing vocabulary, basic descriptions of common test techniques, and the basics of good work habits.

The typical course also teaches a view of testing culture. They teach attitudes about software development and about the motivation and responsibilities of other people who develop software, manage development, or market software. For example, they might include answers to questions like these: How much control should the testing group have over the quality of the software? Should the project manager be able to prioritize the tasks done by the testers? Who should decide what training the testers get? Who should decide how much time testing tasks should take and whether a tester is taking too long? Who should decide what reports the tester should prepare and what data to include or exclude? Under what circumstances should testers refuse to create a report or insist on modifying the assigned task? Who should decide what level of detail and accuracy belong in the specification? What should the testers do if they think this level is insufficient?

Some teachers are very passionate about their view. I am very passionate about my view. My passion does not make me right. The world has many cultures. Each culture offers insights and practices that you would be well served to learn. You can learn about them without expecting them to be the same. Cultural norms will vary from company to company and from industry to industry.

Be cautious about the attitudes that you adopt. As testers, we deliver news about the quality of the product and the progress of the project that is often difficult for someone to hear. That can lead to conflict. You cannot control what other people say to you or what their attitudes are about your type of work, but you can control your evaluation of what they say and how you respond to it. Make your own choices about the nature of conflicts that you will engage in. I think some of the certifications (and not just foundational ones) examine student knowledge at the basic level. They emphasize memorizable knowledge that can be easily graded as correct or incorrect. Questions that can be graded reasonably quickly, the same way by everyone (or by a computer), have no room for matters of judgment. Experts can differ in judgment. As you develop your expertise, you will have to develop your judgment. In doing so, you will have to develop far beyond what is tested by the certification exams, even beyond what I think is being marketed as "expert-level" certification.

Some exam questions also demand cultural/ethical answers that the examiners consider correct. A right-or-wrong-answer exam asks you whether you know what the examiner wants you to answer. It does not ask you whether you have accepted that person's or organization's values as your own. You can adopt your own values and be true to them while demonstrating your knowledge of someone else's culture when you write their exam.
Some testers stay at the basic level all of their career. I think these types of jobs are less secure today than they were fifteen years ago.

Several courses seem designed to make you feel good about yourself and your skills even though your level of knowledge and skill is modest. Learning a few testing tricks or some basic programming or psychology can be very enlightening. Making you feel good is good marketing—for the course providers. Never forget, though, that if what you know can be packed into a few days or weeks of commercial courses, you can be easily replaced by someone who earns much less than you.

How can you go beyond the basics, to add depth?

Going deeper requires specialization. The starting question is what do you want to be good at?
For example, imagine becoming a successful test tool creator. I think you have to learn more about bugs, about the kinds of bugs that you could build a tool to help people find. I think you have to learn more about creating usable, reliable programs that can help other testers (whose skills are weaker than yours) do their work. If your tools support workflow rather than bug hunting, I think you have to learn how to study workflows, how to learn from successful and unsuccessful examples in current use, how to design processes that people will find efficient, effective for their work, and pleasant to use, and how to write code to support those processes.

But imagine instead that you want to test financial applications. You can stay at the surface and hunt GUI bugs and find obvious usability bugs. Below that surface, though, are bugs that will cause people to lose money or to get into finance-related trouble. To go deeper, you have to learn about the type of financial application you are testing. What's difficult in this field? What's confusing? What kinds of things go wrong? If you want to assess the value of a program, you need to know enough about value, in that field, to talk intelligently, and think competently, about it.

You might not want to specialize by application type. You might specialize by problem type. Perhaps you will develop expertise in testing for security flaws or performance problems or video card incompatibility.

Or you might specialize by technique. For example, the master of scenario testing might become an expert in the qualitative research methods used by requirements analysts or human factors analysts. The master risk-based tester might become an expert in digging through the history of this product and related ones, finding ways that products of this kind have gone wrong before, or that the technology used in this product has led to failures in other products.

Each of these illustrates a different way to add depth to your work. This doesn't come just from years of experience. It comes from a deliberate effort to gain knowledge and skill, from intentional practice and intentional searches for relevant information.

People with depth are hard to replace. They know more. They are better at what they do.

Of course, if you specialize in something that becomes irrelevant to the work of your company, you will either (a) have to find a new company or (b) find a new area of expertise.

The nice thing about active, intentional learning is that you don't just learn content. You learn how to learn. Picking up a fourth area of depth will probably be a lot easier and faster than picking up the first.

Over your career, you might develop many areas of depth.

You said that testers should deepen their knowledge and diversify their skills. What does diversifying mean and how will it help them?

Almost all of the best testers I know have worked in several fields, not just testing. They bring many different types of knowledge and skills to their work as testers. For example, along with knowing a lot about testing, they might be skilled in programming, market research, software design, technical writing, etc.

In contrast, when I see a tester or test manager who has been in the field for many years but is not very open to new ideas, or who is constantly fighting with programmers and project managers, I often find out that person has worked in testing all their life. Their perspective is narrow, they don't understand the viewpoints of the people in the other groups that they work with, and their career is often at risk because they haven't kept up with technology.

After you've done testing for a while, I think you should take a break from it. Become a programmer. Or a technical writer. Or a technical support specialist. Do some marketing. Learn how to sell software.

Pay attention to your attitudes and experiences in your new role:

- How do you and your peers look at testers? Why?
- Are you doing the type of work that you used to evaluate when you were a tester? When you were a tester, you had ideas about how this work was done and what made it good or bad. Which of those ideas were completely wrong?
- When testers communicate with you, what works and what really doesn't? Why? How would you change what these testers do, if you could?

Later, when you come back to testing, these experiences will be a foundation for communicating much more effectively.

Over a career, you can do this several times.

Testing is a service. We do tasks for other people. We find information that other people will (we hope) find useful. Every time you strike out in a new direction, you come back to testing with new insight about the ways testing can serve the rest of the organization.

Is better communication the main reason to diversify your skills?

Better communication is one reason. Improving your technical abilities is another reason—an extremely important reason.

Many people who start their careers in testing are weak programmers. This is OK, for a start, but if you want to progress in the field, you should develop strong programming skills. There are some very successful people in our field who are nonprogrammers, but I think it's important to be realistic about what is more likely and what is less likely. I think that people who are good testers and who also have strong programming skills are more likely to earn more money, to find more interesting jobs, to be more secure in their jobs, and to find new jobs more easily. They are more likely to be able to imagine how to use technology to make their work more efficient. They're more likely to be able to work effectively with programmers on agile projects. And there are testing methods, like high volume automated testing, that you simply can't do if you can't program.

Is programming knowledge essential for testing?

Not essential. Some very successful testers are not programmers.

However, when I look at the opportunities that open up for my university students, some require testers to use programming knowledge and others don't. The ones that require programming knowledge are almost always more interesting, at better companies, and they often pay twice as much money.

Million dollar advice, wasn't it? Don't forget to check out part 3 to know what Dr. Cem advises around teaching and coaching testing. Stay tuned!

Happiness is….

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

Like our FACEBOOK page for more of such happiness

https://www.facebook.com/TtimewidTesters

The Power of Geometry in Testing

Geometry is a branch of mathematics concerned with questions of shape, size, relative position of figures, and the properties of space. The shape/form is key to structural strength and architectural aesthetics in static structures. In dynamics, shape/form plays a critical role to higher power transfer/output with lower energy expenditure.

When you are into long distance running (Marathons), 'running form' matters a lot. And this I learnt when I attended a workshop on this year and half ago. The focus was on the erect lean stance with clean forward/backward hand movement. How does this help? Well the erect and lean stance exploits gravity to propel you forward while the rhythmic hand movement enables the core to power you thereby reducing the tiring of legs. So how did it help me? My running performance improved significantly once I got my running form. So what does it take to get into the 'form'? Understanding the science of behind this, then knowledge of techniques of doing and then enormous practice to make this a habit. The result - faster and longer runs, less injury and quicker recovery with no big change to anything else like food/exercise. A significant improvement just by getting in to the 'running form'.

As an avid endurance cyclist I was keen to improve my average speed on long distance rides. Over the last four years I have worked on my stamina, food, strength, mind to increase the distances I could do. Now I was keen to improve my average speed. It was but natural to shift bikes from the comfortable hybrid bike to an endurance race geometry bike, the bike with drop bar handles, sleeker frame. Having

never ridden a road bike, my first ride proved to be marvelous, a 4-5kmph improvement in my average speed. Wow, I had not changed, it was just the bike geometry. And that is when I realized the power of geometry once again in cycling. The bike geometry shifted my stance on the bike and enabled me to transfer more power to the pedals while expending the same energy. And hence the higher average speed. Well the first couple of long rides were interesting, the aggressive geometry resulting in back aches due to the taut back! Once the muscles memorized this stance, it became a habit and the rides are turning out to be zippier. Once again it was about techniques and then practice to make it a habit. These experiences of being more effective (more power) yet be efficient (less energy expenditure and therefore longer distance) in two sports set me thinking on the power of geometry. As a passionate tester, it was but natural to think on this in the context of testing and how this can be applied.

Let us move to testing now…

In both the instances the shape (form) made a big difference to the outcomes. The outcome was faster movement and hence the ability to do longer distances. What is possibly geometry in testing? The arrangement of the system under test (SUT) and the arrangement of test cases to perform a comprehensive evaluation.

What does arrangement of SUT mean? View the SUT as a set of flows consumed by different end users, each flow seen as a composition of business requirements, each one delivered by one/more features with each feature seen as a composition of elemental structural components. How does this 'arrangement (shape)' help? This 'geometry' of the SUT enables better problem decomposition to clearly understand expectations of end users and also 'how it is bolted together'. The arrangement or shape facilitates coming up with intelligent questions to aid deeper understanding of what we know and also what we do not know, the latter capable of becoming defects in the future.

Now let us look at the shape/arrangement of test cases. The arrangement of test cases into groups that target specific types of defects (for a given entity) ordering them in a hierarchical manner over time (levels). Visualize this a multi-layer filter with each layer catching a set of specific type of defects with each layer having variant meshes to catch different types of issues. How could this shape/arrangement of test case help? Well by decomposing the test design activity to making it simpler, sharper and more comprehensive. Not that the earlier thought of arrangement of the SUT enabled questions to understand and uncover potential bugs, while here the arrangement improves the 'filtration capability' enabling the lurking bugs to pop out.

So what does one need to implement this 'geometry' in testing? As with running/cycling, we need to understand the 'science behind' and then understand the techniques, which by practice will become a habit.

Nature uses geometry beautifully. Shapes contribute to strength, result in high utility and ultimately satisfy things via aesthetics/beauty. All this done efficiently! So it is not just about brute force use of intelligence and techniques but about shaping the problem well to understand deeply and evaluate comprehensively.

Look around you. See the various shapes in nature and how they provide the strength and beauty. The honeycomb, the dome of cathedral, the roller coaster, the double helix of DNA… Shape the problem of understanding and evaluation to deliver higher yield. Question deeply, understand better and evaluate comprehensively.

Exploit 'The power of geometry in testing'.

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

Passionate about excellence, his mission is to invent technologies to deliver "clean software".

He can be reached at **ash@stagsoftware.com**

Back To Index

testing intelligence

- *its all about becoming an intelligent tester*

an exclusive series by **Joel Montvelisky**

Decision Driven Test Management

6 tips to improve the value of your testing

You were not hired to find bugs!

I have said this a number of times in the past, if your management thinks that your job is to find all the bugs in the product and deliver a defect-free release at the end of the process, I strongly recommend you find another management…

If you are lucky and work on a smart company, then the main objective of your work as a tester and especially as a test manager will be better described as follows:

To provide stakeholders with visibility
into the status of the product and process,
so they can make the correct decisions.

Don't get me wrong, we are still testing and reporting on our findings, but in this definition the focus is not on the product or even on the testing, it is placed on the stakeholders and the decision they need to make (based on the information we provide them).

In order to explain this test management approach better I want to start by giving it a name:

Decision Driven Test Management

Many experienced test managers already use DDTM or Decision Driven Test Management unconsciously in one level or another. In lots of ways it is the logical way to work, and many of us adopt this approach even without noticing it.

But I don't recall someone defining it explicitly or teaching it publicly yet, and so what I want to do is to help understand how we can all use this approach consciously to improve the value of our test management work, and to teach it to those who are still moving towards this methodology but have not yet made the jump.

In a nutshell: Start from the people and their decisions, then plan your tests accordingly

Most people assume that when you start a testing project you begin by learning the product, then you plan the tests you want to run, and then you create a work plan or schedule.

Well… this is wrong!

The truth is that most projects usually set their release dates and internal milestones long before they start thinking about testing.

And so you will usually see an experienced test manager start a project by learning and understanding these dates and milestones, only then will he or she move forward to learn the product, and then based on the understanding of the complete situation will he or she plan the work schedule.

The reason we look at the project plan first is that by learning our milestones we also learn a lot about the decisions that need to be made as part of the process, and so we are starting by understanding what information will be required from us and when will this information be needed, and then we can plan our testing accordingly.

6 tips to master DDTM

It is not always easy to grasp the small things that experienced managers do without even noticing, so let me try and explain the simple yet important principles behind this approach to test planning and test management.

What are the most important things you need to do to work correctly and efficiently with DDTM?

1. **Map your stakeholders and listen carefully to their needs.**

Start by understanding who you are working for – and no, it is not (mainly) the end user!

Make a list of your project stakeholders and then prioritize this list based on who is more important to the project and to you. Then go and talk to them to understand what they need to know as part of the project.

Typical stakeholders are Product, Project and Release Managers; Development Leads and even Developers; sometimes your circle will be broader and it may include VP's of Marketing, Services, Supports and even Sales; and in some occasions when the project is very important or when the company is relatively small it can even include the CEO.

Each time you will have different stakeholders and you will need to make an effort to find all of them or at least the most important ones.

From my experience, the biggest challenge is that these people are not really aware of what information they need, so you will need to work with them to define these needs.

2. **Plan tests and deliverables based on the information needs.**

Take the information needs of your stakeholders and translate this into concrete information deliverables. You are looking here to plan your Metrics, Reports, Dashboards, etc.

Schedule these deliverables based on when they will be needed, many times these "delivery dates" are the milestones already defined in your project.

This will give you the internal milestone plan for your testing team.

3. Plan your tests according to your information needs, understand what information you want to capture up-front.

Now you know what reports, dashboards, statistics and additional information you need to provide and when. This is what you need to plan your testing operations.

Make sure to explain to your testers why they are testing and what information to look for. Many times we feel that our junior testers do not care why they are running the tests we give them, and this is one of the worst mistakes to make.

Your testers are intelligent (and if they are not then replace them!) so make them part of your "Testing Intelligence Team" and help them bring forward the information that will help your team make the right decisions.
Sometimes these findings cannot be planned, and they will depend on the avid eye of a smart tester to be found and reported in time!

3. **Be ready to change and improvise.**

Fact No. 1: You will not have enough time to test everything you need to test in order to provide the information that is required.

Fact No. 2: Even if you manage to plan your schedule perfectly to fit every need known at the beginning of the project. As your project progresses things will change and more stakeholders will require more information from you.

If this is the case, then you need to keep your eyes on the ball all the time, and don't lose your mind when you realize that people have new questions, and that you will need to alter and improvise your plans in order to keep up.

People are not bad because they change their minds or modify their requests!

Change is the only constant parameter in most projects Embrace change, you have no other alternative.

5. Stream information via multiple channels.

Here are 2 additional and important things to remember:

– Different people absorb data differently.
– Sometimes you will need to present the same information twice or three times before it is absorbed and understood by your busy stakeholders.

It is more effective to use multiple channels to stream your information, and so help your goal of supporting the decision-making process.

Use Dashboards, Kitchen Monitors, Email Reports, Meetings, etc. to pass along the information needs of your stakeholders.

6. Make this work iterative, improve with each iteration.

The first time you try this approach you will fail poorly!

The second time you will feel that you were almost able to help but not there 100%…

The third time you will start seeing a difference in the approach of your company and stakeholders towards your testing. They will begin noticing that you are proactively coming with the information they need.

From then on, they will come to you and ask you to be a more active part of the decision making circles.

This type of project, especially at the beginning, will be a continuous improvement effort. Knowing this may help you cope with it better

*** Bonus tip – remember you are providing a service.**

For some reason some of my best testers have been people who worked previously either selling things or waiting tables, really!!

Why? I think this is because they understand that they need to provide a service and so the "customer" is always the most important thing in their work.

I love geeks and computers, they are some of my best friends in life but when you work in testing you need to have soft skills and not only analytical skills.

Do you use DDTM in your process? Share your tips!
As I mentioned before, I am sure that many of you already work this way without even noticing it.

If you do, please go ahead and share with us your experience! What works best, and what do you do to make it better in your team?

Joel Montvelisky is a tester and test manager with over 14 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

Back To Index

testomaton

Quality Assurance via Automation

Software testing startup specializing in test automation services, consulting & training for QA teams on how to build and evolve automation testing suites.

Services

Test System Analysis and work estimation

Review of client's system (either in development or on early stage) to produce a Test Plan document with needed test cases and scenarios for automation testing.

Tests creation and Automation

Development of test cases (in a test plan) and Test Scripts to execute the test cases.

Regression and Test evolution

Development of Regression test suites and New Features suites, including test plans and test scripts or maintenance of existing.

Architecture Consulting

Review of software architecture, components and integration with other systems (if applicable).

Trainings

Depending on the particular need, we can provide training services for: Quality Assurance theory and best practices, Java, Spring, Continuous Integration, Groovy, Selenium and frameworks for QA. For further information please check our web site.



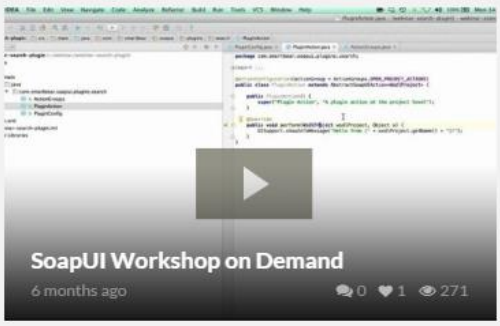
We enjoy putting our experience to your service. Our know-how allows us to provide consultation services for projects at any stage, from small up to super-large. Due to our range of expertise we can assist in software architecture and COTS selection; review of software designs; creation of testing suites and test plans with focus on automation; help building quality assurance teams via our extensive training curriculum.

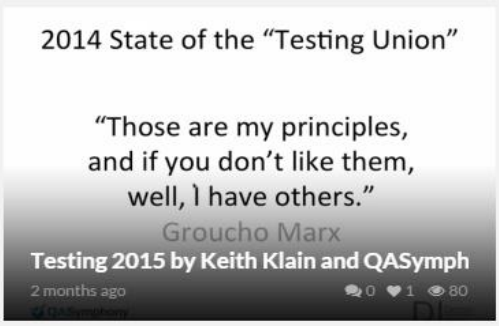
look us up: www.testomaton.com - twitter: @testomaton

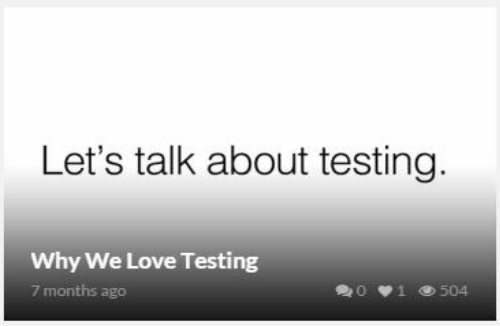
Got tired of reading? No problem! Start watching awesome testing videos…

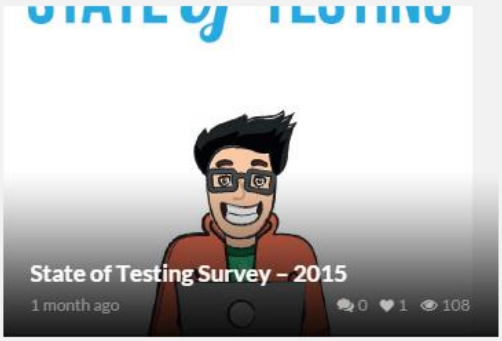
TV for Testers

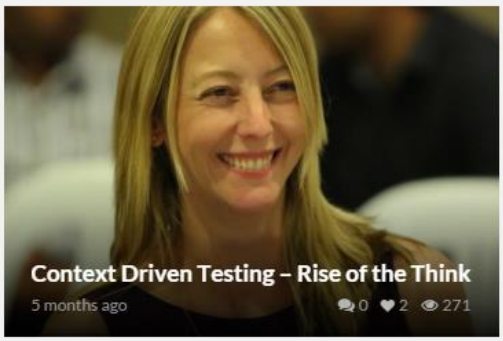
Your one stop shop for all software testing videos

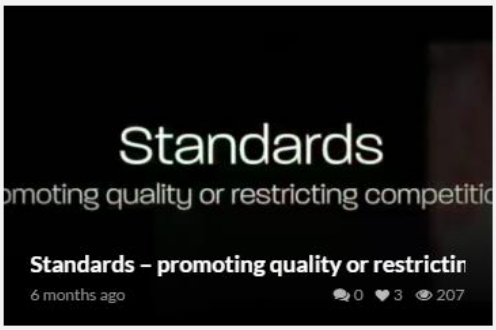












WWW.TVFORTESTERS.COM

www.talesoftesting.com

What does it take to produce monthly issues of a most read testing magazine? What makes those interviews and articles a special choice of our editor? Some stories are not often talked about...otherwise....! Visit to find out about everything that makes you curious about **Tea-time with Testers!**

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.

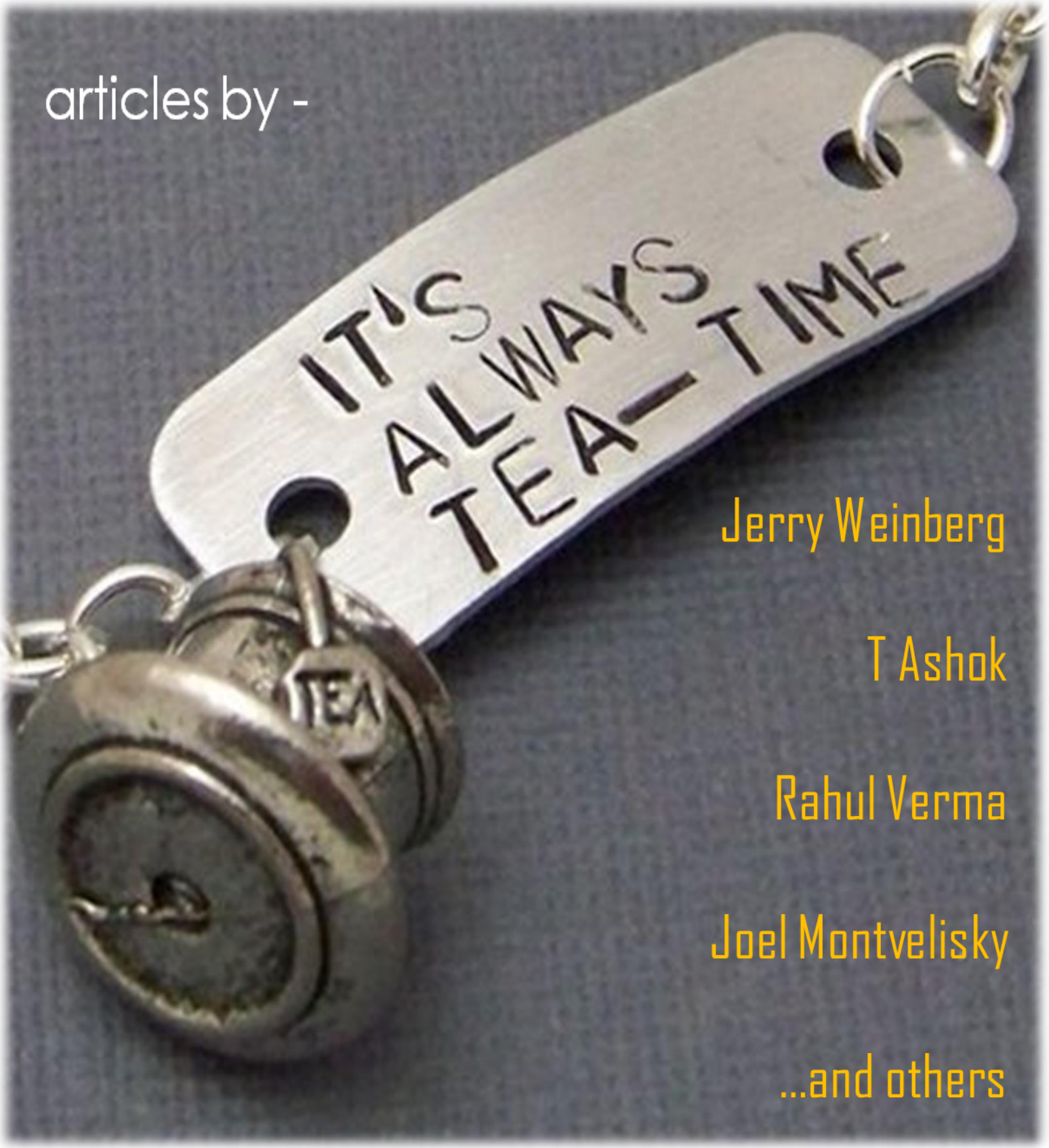
And the **Good News** is…

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 – Dramageddon

Core Team:

Dr.Meeta Prakash (Bangalore, India)

Unmesh Gundecha (Pune,India)

Dr. Meeta Prakash

Unmesh Gundecha

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