Musings over Tea-time

Anthology of T-Talks

T Ashok

Celebrating THREE Years of Tea-time with Testers

ANNIVERSARY SPECIAL

Foreword by Editor, Tea-time with Testers

It was lovely winter of year 2010. I still remember that morning when I was going to attend a very first testing conference of my life. A full day testing conference with lunch and that too at participation fees of Rs. 50? "Wow!", I said to myself and thanked organisers at Silicon India for making it happen.

Thanks to Mumbai's traffic, I was late for the conference. The auditorium was full and a gentleman with well shaped beard, in grey blazer with a refreshing smile commenced his talk.

His voice was sharp and so was his sight, and what should I say about his ideas? They were clean, smooth, and he was presenting them with his brilliant analogies. For the very first time I got to know that aesthetics and ergonomics can apply to software testing too and this man made me fall in love with the craft of Software Testing.

Who else that man can be? I don't know how to thank T Ashok, for his talk left an everlasting impression on me. He made me (and many like myself present there) think about testing, our daily job in much different way and it is probably the result of inspiration I took from him, that I could create a forum like Tea-time with Testers.

If you are regular reader of Tea-time with Testers, you would notice that he has been tirelessly writing fresh articles each month, from very first issue of this magazine. There is something unique about his style, be it the way he presents on stage or the way he

writes about testing. I am big fan of Ashok's writing and it won't be exaggeration if I call him the master of analogy. Pick any article from this book and you'll realise what makes me say so.

Like some other awesome books on testing, I personally feel that this book will gain special place in your mind as it touches variety of subjects around software testing. Each chapter is special and provides practical solution to real time testing problems. You don't need to read this book in any particular order. Just pick any chapter and I promise, you won't realise when you started reading next one. Each article chosen for this book is interesting. His ideas are simple yet logical and powerful. I can't admire him enough for the ease with which he presents his ideas and explains them with equally fantastic analogies. If I have ever loved reading testing stories written by someone else after Jerry Weinberg, I would proudly say that those are Ashok's stories. With his stories, he has made me believe that a guy named Joe really exists somewhere. Do not miss his stories 'The promising athlete', 'The diagnosis' and 'The tale of two doctors' which are included in this book.

I must tell you that T-talks column in 'Tea-time with Testers' where Ashok regularly writes his thoughts is very very famous among our readers. So much so that, our friend Kiran from Quality Testing celebrated his birthday by distributing printouts of Ashok's article (Test Case Potency) to his team. Isn't that awesome way to celebrate tester's birthday? And what other example should I give to explain popularity of this column?

This book, which is an anthology of T-Talks column is pure bliss for those who love reading about testing. Be assured that you are getting to read the best of T Ashok, all at one place, here.

I had never imagined that my new journey that started with Ashok's talk at Silicon India would offer me opportunity to work with him so closely. I can't thank him enough for his tireless contribution to Tea-time with Testers and to software testing community as a whole. This book is our small and humble attempt to appreciate his selfless contribution.

Enjoy your tea time and share the happiness with your friends!

Lalitkumar Bhamare
Editor and Co-founder
Tea-time with Testers Magazine

Author's note

A delightful interaction with Lalit three years ago gave birth to a column titled T-Talks in the new eZine Tea-time with Testers. And it has been a wonderful journey, to come up with something interesting every month.

It has been a journey of thinking, observing, reflecting, being inspired from the numerous daily interactions and correlating these to testing. The joy is in the simplicity of these. I have arranged these musings into three sections (similar to my journey) - "Think & Do", "Observe & Learn" and "Realise & Evolve".

I am delighted that Tea-time with Testers is 1000+ days young. Heartiest congratulations to Lalit. I am happy to be part of these celebrations with this anthology.

Now pick up a good cuppa tea and Enjoy!

Section 1 – Think & Do

This section contains articles that espouse scientific thinking is the cornerstone to doing a job well. Enabling the understanding that testing is an intellectual act. And logical thinking enables you to see clarity in the problem space and therefore test well.

Scientific techniques that aid in problem decomposition, understanding, communication, design, questioning, optimisation, complexity analysis, clarity enablement, test quality assessment, automation, clarity in terminologies, are what these articles highlight.

Scientific thinking.
Do well.

Think better using "Descriptive-Prescriptive" approach

Nov 2013

When you want to understand, 'storify'. Describe.
When you want to solve a problem, state rules. Prescribe.

Testing is interesting as it is unbounded. Customer expectations constantly expand, overall development effort/time is expected to shrink and quality is expected to be better than before. And this requires good (problem) analysis and (solution) synthesis skills. What does it take to analyse a problem and to synthesise a solution? Let us think differently...

The prerequisite to good problem analysis is a clear and deep understanding of the problem. Now how do we understand something well? Remember when you were young, you were told stories to help you understand good/evil, right/wrong etc. Story telling aids understanding. Story telling is describing something in a interesting fashion, of connecting various elements in an engaging manner, with a human touch. And that is possibly why we relate to a story better rather than mere facts enabling good understanding and fostering interesting questions to deepen the same. Describing the elements and connecting these enables us to come up with interesting questions and attempting to answer these helps us to understand better. This is what I term as 'descriptive approach'.

Now let us shift to how we possibly discover solutions to problems. Now what is a solution? A set of rules to follow to solve. A 'prescription' that we can follow. Synthesising a solution requires us to understand the various conditions that relate to the problem and therefore a combination of these conditions is the

'prescription' to follow. Focussing on a 'prescriptive approach' enables us to extract conditions and formulate the solution. The synthesised solution is expected to be clear, unambiguous, precise i.e 'objective' while on the other hand problem analysis is aided by a descriptive approach which is 'subjective'.

Now let us connect these... Problem analysis requires a 'descriptive approach' while Solution synthesis requires a 'prescriptive approach'. The former is subjective and interesting involving story-telling, while the latter is objective and cold, to extract conditions. And these conditions connected to form 'prescriptions' to follow, to solve the problem.

Where are we going with this? How does this relate to testing? Hmm. Let us look three parts in the test lifecycle...

Let us look at how we understand a product/application and formulate a good baseline for testing. We attempt to understand a system by reading the specification or 'playing' with a prior version of the system. The act of writing 'user stories' is to describe as what the system should do as a simple story. A typical specification could be dry and boring, the trick is to attach a human touch i.e who (person or another system). The key aspects that we connect are who uses this, why do they want it, what do they value, when do they use it, how frequently do they use it, how do they intend use it. Keeping the story telling angle, and starting from the 'who' and looking at the various elements allows us to describe the system and therefore come up with questions. So when you want to understand 'storify'!

A prescriptive approach to solving the problem of understanding a system could be to see the system as collection of specific

information elements that need to be connected. And the process of connecting the various elements that relate who, what, when, how, why. Think of this a small mind map to describe the system or part thereof, where we connect various elements like : various type of users, when and how much they use, how much they value this, attributes expected, environment that it needs to be supported, the state of development (new, modified), conditions that govern the behaviour, the list of features/requirements that it is made up of, the deployment environment. The process of connecting these various elements using a mind map like approach is a rational way to decompose and understand, a 'prescriptive approach'. This is embodied as a technique called Landscaping in Hypothesis Based Testing (HBT).

Ultimately stating the baseline to test as list of features/ requirements/flows with attributes to satisfy is clean 'prescriptive way' to state 'what to test' and 'test for what'.

Let's move on to test design....A earlier article (TTWT Sep 2012 : "Behave yourself - Descriptive to Prescriptive") described how this is applied in test design. The design problem is one of extracting the various conditions that govern the intended behaviour. Describing how it works/intended-to-work allows to understand and identify the conditions - 'descriptive approach'. Once we identify the conditions the solution of designing test scenarios is of 'stringing the conditions' to result in scenarios to evaluate i.e prescription. So problem of design requires 'story-fication' of behaviour conditions, a descriptive approach, while the solution to designing scenarios requires prescribing the behavioural model as a combination of conditions. Note that this is applicable to design of functional or non-functional test scenarios.

Now lastly let us discuss how this thinking approach can be applied to reporting and management. When we report progress, quality or delivery risk, it is common to report using metaphors like charts, metrics. When somebody reports dry numbers, charts, my instant reaction is "what does this mean?" and "is this good or bad?". If the metaphors i.e charts/metrics are crisply described, then we could relate to it better and understand the status clearly. Once the status is understood well, I can compare with limits/ benchmarks ('prescriptive') to formulate action plans to resolve problems.

So think better using 'descriptive approach' to analyse a problem and apply a 'prescriptive approach' to synthesise the solution. This approach forms the backbone of HBT with set of 'prescriptive' techniques and syntax of writing to aid better 'description' to enable good understanding.

Describing is very warm and human, while prescribing is a cold and machine-like! So the next time you encounter you a problem, apply the descriptive and prescriptive approach. May the heat subside!

This is the wonderful season of the year when all the problems of the year fade into a hope for a great new year. Have a lovely December. Merry Christmas and Happy Holidays!

Language shapes the way we think

July 2013

Syntax is a great guide. A guide who provides you the rules.
Rules that enable you to stay on the path of clarity.

Language is not just for writing, it plays a significant part in our thinking process. It has been discussed widely as how "Language shapes the way we think" (Read and listen to one of these at http://blog.ted.com/2013/02/19/5-examples-of-how-the-languages-we-speak-can-affect-the-way-we-think/ for an interesting TED talk blog & talk).

Once we are comfortable with a language, we "think" in that language. For example, we form sentences in the mind to understand, form narratives to describe, think in terms of if-then to discern logic, create command sequences to create actions and so on.

Language is made of the syntax 'the rules' and the content 'the semantics'. The syntax i.e rules shapes the way and the depth of understanding of the content.
Language allows us to:

1. Describe a story "Understand
2. Breakdown the problem "Simplify"
3. Setup clear boundaries "Baseline"
4. State the purpose "Goal"
5. Organize our thoughts "Plan"
6. Issue instructions to get things done "Action"
7. State what has happened "Report"
8. Document stuff so as not to forget "Remember".

Now let relate these to 'testing'. Look at the 1-7. Sounds familiar? This is what we do when we commence testing - Understand, Simplify, Baseline, Setup goal, Plan, Action/do(Test), Report and Remember for future.

Now let us see how the language which we typically use to document/write shapes how we think.

1. Describe a story "Understand"

Understanding is a key element to good testing. To understand whose need(s) we are trying to satisfy and the value expected, it is critical to think like an end-user. We often use the term "think from the user point of view", but it is easier said than practiced. To enable a deeper and better understanding of the system a persona-based approach i.e of Describing the behaviour and the associated attribute(s) in first person as if the end user is describing it from his/her point of view enables you to put yourself in the shoes of the end user, empathise and therefore understand better.

2. Breakdown the problem "Simplify"

Any non-trivial thing is presumed complex. A true hallmark of good understanding is de-mystification, that of making it simple. From a language perspective, it is about summarizing, of describing in short sentences and not exceeding a paragraph.

3. Setup clear boundaries “Baseline"

A clear baseline as to what-to-test (I.e requirements/features) is necessary to ensure clarity in what we need to and that we have indeed covered i.e evaluated completely. Using a imperative style sentence that is short and precise forces us establish a clear baseline.

For example a customer requirement may be stated as “That the system admin shall be able to ...” while an example of technical feature is "That the system shall provide ....".
Note that a descriptive or a narrative style is a strict no-no here.

4. State the purpose "Goal"
Testing is about ensuring that the needs of the various end users delivered via the technical features do meet their expectations. Not only it is necessary to clearly outline the needs as a clear baseline, it is equally necessary to ensure that the expectations are well stated.

This implies that the baseline has to be qualified with a criteria that is indeed objective. For example “That the system admin shall be able to do 'blah' within 'x' seconds on these 'y' devices”.
I.e A short imperative sentence with a qualifier that is objective.

5. Organize our thoughts "Plan"
This is one of the things that we do most frequently in daily life, the To-do list. The way we do this is to list down activities in a numbered bulleted list in sequential order (based on time).
The language that we typically use is in first person using an imperative style. The method that we use to think is in terms of bullets with a imperative style heading with a narrative style to describe the plan of each To-do action in detail.

6. Issue instructions to get things done "Action"
This is where we come up with scenarios to test. What I have observed is most often a narrative style description that describes the actions to performed, the data to be used, and the method of validations to assess correctness.

From a language perspective it is necessary to be action oriented here I.e describe each scenario as a command and the associated expected result in single sentence and then describe the steps to perform. For example "Ensure that the system does/does-not 'foo' when 'bar' is done". First be clear of what you want to accomplish before you jump to how-to-do.

7. State what has happened "Report"
Now this is the fun part as reporting can describe multiple facts that are all connected, leading to complexity and confusion. From a language perspective, reporting is describing outcomes arranged by elements across time with associated detail and therefore the sentence to describe these can turnout to be inherently complex. This is applicable to defect reporting, reporting test cycle outcomes, to reporting final rest results and to describe learnings etc.

To ensure clarity of thought, it is necessary to partition the description first in terms of summary and detail, then partition the detail into smaller elements, describing each element along various dimensions.

In the case of defect report we describe a short synopsis of the problem and then then describe with multiple elements like 'detailed observation', 'method to reproduce', 'environments observed in' etc.
In the case of test report, we commence with a summary and then describe the various each element in a section with different dimensions to describe in detail the elemental information as possible subsections.

8. Document stuff so as not to forget "Remember".

This is really the free form part, the part that we jot down everything we observe, learn from past. This is one part that we cannot stick to one style of syntax. This is a mixture of all styles mentioned above and beautiful mixture of terseness with detail.

The structure of sentence matters to the way we think, understand, perceive. Ultimately the content(semantics) matters, but the structure does matters too. Syntax is a great guide. A guide that shapes how you think, enabling you to stay on the path of clarity. Syntax used in a rote matter may be seen as restrictive, but clearly it marks the path of clarity. Use it. Use it wisely.

It matters how you write/document. Clarity is truly a function how you describe. Remember language shapes the way you think or how it makes others think!
Enjoy!

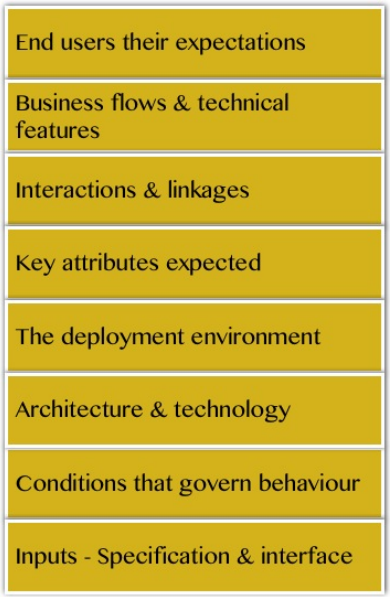
The onion peel and the half-filled tea cup

April-May 2013

Good understanding is not about knowing more.
It is about discarding what is not necessary.

Good understanding is critical to good testing. So how does one understand a system well quickly? The typical belief is that prior experience and domain knowledge play a key role here. But what if we do not have sufficient prior experience in that area? That is when when we like to resort to creative thinking, good questioning as some of the means to understand. Now comes the question - "how does one come up with good questions" to understand better? In my discussions with folks in the community, the typical answer I get is that this is based on prior experience. Now we are back to square one- Experience! My take is that there is a logical/ scientific way to go about this and therefore need not be driven only by experience/domain-knowledge. Let us discuss this in detail...

The act of understanding is interesting! It is immersive, non-linear, instinctive, frustrating yet fun, detailed but not forgetting the big picture, time constrained and extremely satisfying when we "get it". To look at the act of understanding in a scientific manner, it is necessary to be clear as what we want to understand and when. How do we onion-peel?



Using the picture alongside, Let us start from the "outer layer" of the onion peel (top). It commences with understanding who the consumers/ end-users of the system are and how the system is intended to help them, what they may expect of the system. Thinking from the end user perspective, the focus is to understand what each of them do/ intend-to-do with system and build the baseline of the use-case(s) / business flow(s).

Now we are ready to identify the technical features that enable these business flows to be accomplished and then setup the technical feature baseline. Having broken down the system into constituent flows/ features, understand their "connections" I.e how does each flow/feature depend on other and therefore their linkages.

Before we get deeper into understanding in detail how each flow/ feature works, understand the key attributes of each of them. Note that we have decomposed the system and setup a clear baseline of the system in terms of users, their business flows, constituent technical features and the associated attributes. Now we are ready to delve into the details. Did you notice that we have minimised our dependency on domain knowledge ?

Let us peel the onion further... Before we get into the detailed understanding of each flow/feature, understand how the system is deployed. What other systems does this interact/use, how is it

deployed, what is the environment in terms of hardware, software, versions this needs to support/uses. Having understood the external environment, delve next into the internal structure - what are the various constituents of the system, how are they interconnected, the interfaces and the technology(ies) used to construct these. What we have done is to get a good feel of the external environment and the internal structure. Now we are ready to understand the behaviour of each flow/feature.

What is behaviour? It is about about transforming the input(s) to appropriate output(s) i.e. to outputs based on certain conditions. So all we need to do is understand for each flow/feature, the inputs and the outputs, their specification and the conditions that constitute the business logic. Finally we need to get understand how these inputs enter the system i.e input interface.

Good understanding is not about knowing everything in detail, it is about just knowing what is required and learning to discard what is not necessary or deferring to a later stage.

I am reminded of a Buddha story, listen to this.
Once upon a time there lived a wise and learned man who had mastered various religions, philosophy. He was proud of his knowledge and wanted to learn more. He came to know that Buddha who lived in another state was a very learned man too. He was keen to further his knowledge. went to Buddha and introduced himself as a very well read and learned man who knew all the systems of religions & philosophies. He expressed his wish to enhance his knowledge and requested Buddha to teach him. Buddha said he would be happy to do so and requested him to kindly sit in the corner of the room. A few hours passed and the learned man was getting fidgety while Buddha continued to do his

work, and as the sun went down Buddha requested that he will do so next day. The man went away only to be back next morning, Buddha requested him sit down in the corner. The morning turned to afternoon and dusk was setting in. The man was throughly upset that Buddha was ignoring him, he went up to him and said “I am upset that you are ignoring me. I hope you understand that I am a learned man and do posses a lot of knowledge in the matters of religion, philosophy”. Buddha was expecting this and he said “Come, sit down let us have tea”. Buddha took the tea pot and poured the tea onto the cup while watching the man. The cup was filled to the brim and Buddha continued pouring, spilling the tea. The man then shouted “The tea cup is full, stop pouring!”. Buddha said “Your mind is like this, already full (of knowledge) with no space to acquire additional knowledge”. Empty it and you will ready to learn.

Good understanding requires to ensure that tea cup is only partially full i.e know only what you require for now, defer what is not required so that you do not just know, you understand. It is not attempting to knowing all.

Before you jump into testing anything, Understand. Don’t be fazed by complexity nor be worried by no prior knowledge. Apply the onion peel keeping the tea-cup half filled.

Enjoy!

Too many conditions!

January 2013

With no rules, comes immense freedom and power.
And with great power comes responsibility.

The coming of new year is always interesting. It is the time of the year when new resolutions are made. The word 'resolve' indicates grit, something we strongly wish to comply with.

What does this have to do with software testing/quality ? As a software developer, at the start, we make a resolution to deliver great quality software. In real life the new year resolutions are quietly forgotten as days/weeks pass by. But not in the case of software. It is necessary to meet this and requires effort from developer and tester.

What does it take to accomplish this? (I.e fulfill the resolution) It requires one to comply with certain conditions and ensure that they are not violated, thereby exhibiting the new desired behaviours. Non violation of the identified conditions is very necessary to demonstrate 'grit' and thereby meet the resolution.

To deliver great quality software, it requires identification of the various conditions that ensures:

1. Behaviours are as desired when the conditions are met
2. Unexpected behaviour is not exhibited when conditions are violated.

Sounds familiar? Of course yes! Test scenarios are really combinations of conditions.

Let's examine the conditions in detail... There are a variety of conditions- they pertain to data, interface, (internal) structure, functional behaviour, (external) environment, resource consumption, linkages to other systems, and other conditions to the system attributes (non-functional aspects).

Enumerating this ...

1. Data related conditions: Data types, boundaries, value conditions
2. .Data interface conditions: Mandatory-ness, order, dependency, presentation
3. Structural conditions: Linkages, resource use policy, timing, concurrency
4. Behavioural conditions: That which governs the functionality, the business logic
5. Flow conditions: The larger behaviour, business logic of end-to-end flow
6. Environment related: Messing up or being messed up by the external environment
7. Attribute related: Load conditions, performance conditions, security conditions etc
8. Linkages to other systems: Deployment conditions

Ultimately testing is about assessing that the behaviour is as desired when all the conditions are combined. Now we have 'Too many conditions' ! Now meaningfully pare down the complexity by partitioning. In HBT (Hypothesis Based Testing), this is accomplished by setting up NINE Quality Levels, where each level focuses on certain conditions and their combination. Note the the EIGHT sets of conditions that was described earlier map to the Quality Level ONE through EIGHT.

Having partitioned thus, it definitely becomes much easier to combine a smaller set of conditions at each level and ensure compliance and non-violation. Thus the chances of meeting the resolution is much higher.

So when we test software, appreciate that all we are doing to is check the 'compliance to' and the non-violation of' combinations of conditions. And to ensure that we are clear if what we want to do, partition these into Quality levels, where only a smaller subset of conditions needs to be combined. So test scenarios are generated at each quality level, and these are complete, smaller and manageable !

On a personal note, my new year resolution is to be a super randonneur and do a 1000km brevet. So many conditions need to be met to accomplish this - endurance, mental toughness, sleep management, climbing, environment resilience. Instead of attempting to combine all at one ago, I have partitioned these and combined a smaller set of conditions and have met with success in the first month.

So what is your new year resolution? Do not give up or forget! Identify the 'Too many conditions' and break it down and comply. All the very best.

Au revoir.

How do you solve a problem ?

November 2012

Logic dissects a problem from outside and then solve it.
Experience helps you understand the problem from inside and then solve it.

Every moment in life is an interesting one, as we encounter new problems that we are challenged to solve. Some of these are ones that we have encountered before and therefore we know the solution, while some are new for which we have to a figure out the solution. But, how do you solve any problem ?

Well the easy answer is "Based on experience". You have encountered the problem before, solved it and therefore have to apply the same or a modified solution to the current problem. Sounds familiar ? A common question that we encounter - "Do you have the relevant experience in this domain/technology to test this software?". The premise is if I have tested similar systems, I should be able to do a decent job with the current system.

The other answer - "Based on sound logic/technique". I have not solved it before, but I know the algorithm, the technique , the logic to solve it. A great answer, but people are skeptical, unless you are "certified" in the application of it. Techniques are the result of scientific thinking, the application of logic.

Now what do these imply? In the former case, you need to have experience, this takes time and is expensive, while the latter is cheaper as it can be taught. Are there any other ways ? Yes, in some cases, we are taught certain principles, that we apply. Principles are not exact techniques(formula), but are conditions that

we use to make choices to solve a problem. Consider this: if you are walking eastward and the shadow is behind you implies that it is still morning, while the shadow in the front implies evening. So if a question (I.e. a problem) was posed to you to figure out the time of the day given the shadow position and the direction of walking, the “Principle of Shadow” helps you figure the answer.

In some cases, we may not have have a exact formula or a clear set of conditions, but a set of directions to choose from based on some information. These are "Guidelines", that lists different situations and suggests what to do in each case. For example a guideline may enable you choose a test technique based on the type of fault you wish to uncover. In the case of complex logic, use code coverage techniques, in case of complex behavioral conditions, use decision table.

Let us attempt to create picture of this..
Problem solving based purely on experience is based on the "individual’s skill", whereas an approach based on the logical/ scientific thinking depends on the "strength of the process". Skill-based problem solving can be seen as an 'art/craft while a logical approach can be deemed as one based on science & engineering i.e. scientific principles + process of usage of these principles.

The figure alongside depicts the problem solving approach as being dependent on these two aspects - "People Skill" and "Process Strength".

People skill	Process Strength	Problem solving approach	
C	A	Technique	
B	B	Principle	Science & Engineering
A	C	Guideline	
A+	C-	Pure experience	Craftsmanship

Note:

1. People skills are rated as A+ through C to indicate the individual skillfulness needed from : Highly Experienced/Skillful to Least
2. Process strength are rated as A+ through C to indicate how strong the process/technology is : From Very Strong to Least Strong

The scientific approach to problem solving can be:

1. A formula-like approach, algorithmic titled "Technique"
2. Decision enabler that outlines key points to make-choices/ choose-path titled "Principle"
3. A broad brush approach that is suggestive of situations and what to to then titled "Guideline".

The ideal approach would be the one that is based on "Technique", so that an individual can predictably apply rather than depend solely on experience. Note that acquiring the experience does take time, and this cannot be shortened drastically.

The crux of HBT (Hypothesis Based Testing) is based on a set of disciplines of thinking that consist of tools based on these three problem solving approaches - Technique, Principle and Guideline.

The fun part of being an engineer is encountering problems and devising solutions. So the next time you solve a problem, identify what the approach was : Did you discover/apply a 'Technique', 'Principle or a 'Guideline'? This will greatly help to build and refine

your problem solving toolbox. And it is a wonderful feeling to have a greats toolbox. A big SwissKnife to solve all problems.

Well, what is the approach to solve problems that I encounter with my spouse? None of the above! "Accept whatever she says" and the problem is solved !!

Well until next time, have fun. CIAO.

Behave yourself !

Sep 2012

Good behaviour is about compliance to rules.
Good test cases find loophole(s) in these rules.

One of the common statements that I come across is "Domain knowledge is critical to testing functionality". The belief is that knowledge of the domain acquired from prior experience is very necessary to perform good functional testing. Not that I disagree, because prior experience is always handy. But the key question is 'What if I am encountering a functionality for the first time? Would I be effective?'. Wouldn't it be nice to be able to do this scientifically using some techniques?

Let us first examine the basics... The objective of functionality testing is to ascertain if the behaviour of the entity under test (EUT) is as intended. So what is behaviour? When we tell a unruly kid "Behave yourself", what we mean is 'please comply with the acceptable norms i.e. rules'. Hence behaviour of an EUT is about 'obeying some rules'. These 'rules' are really the specification. So to test a functionality of any EUT, we need to first understand the intended rules and then create scenarios that are really various behaviours by combining the rules in different ways and then check the outcome(s).

So how do we figure out the rules? Hmm... If I had domain knowledge then I would know how this should work, and therefore the rules. Note that this can be dangerous too, as this thinking sets up a bias, as we may assume certain rules that may be incorrect. Anyways, the issue is, how otherwise? To understand the

intended behaviour, first describe what the EUT is supposed to do. Describe the behaviour as a series of steps. Each step accepts/ uses data and processes the data according to some condition(s). Examine each step and extract the condition(s) hidden in the step. Voila... We now have the various conditions aka rules.

This technique of extracting conditions (aka rules) is what I call "Descriptive to Prescriptive", that is converting the descriptive specification to a prescriptive specification. "Describing" enables us to understand the behaviour from first principles while "Prescribing" helps us to validate the behaviour. We all know the act of clear and comprehensive description consists of the 5W+1H - Who, What, When, Where,Why & How. So when you encounter a functionality for the first time, apply 5W1H and describe the behaviour as a series of simple steps.

Now examine each step and identify conditions implicitly or explicitly present. Once the conditions are extracted from each step, re-write each step as a series of rules and what we have now is prescriptive behaviour. From this it is easier to create the test scenarios and then identify the various data sets for each scenario and subsequently generate test cases and therefore for the entire EUT.

This behaviour based approach to functionality testing is implemented by two techniques 'Box Model' and 'Behaviour-Stimuli Approach' in HBT (Hypothesis Based Testing). This lessens the need for domain knowledge as a prerequisite by enabling one to question better to understand behaviour, and then extracting conditions to test effectively.

So whether you test a technical feature or a business flow try this approach. Be the doctor - “Ask the patient to describe so that can you can prescribe”. Enjoy the happiness that comes out of the cure.

May you contribute to healthy software and a cleaner world.
Have a great day.

Agile Sutra – "Sensitize & Prevent", not "Design & Execute"

Aug 2012

Write less. Communicate more.
Think deeply. May the light flow into you.

As a consultant deploying HBT (Hypothesis Based Testing) in Agile environment, I made an interesting discovery. That the focus of testing here should be to "sensitise & prevent" defects rather than "design & execute" test cases.

In the Agile environment, the focus is on decomposing the problem into small user stories and delivering it. This implies means that we are decomposing complexity and thereby demystifying it. By deduction therefore, the code should be more correct than wrong. Also because the element of delivery 'an user story' is small, it should be easy to test and therefore easy to convert the test into script. And therefore testing is interwoven naturally with coding. Does it mean that the code is cleaner?

In my interactions with the team, I discovered that despite the code being delivered faster, quality challenges exist. Customer reported defects still keep the team busy. Challenges because of extreme focus on the 'small' and not on the larger picture??

Let me illustrate this.
Situation #1
The user story in point is a logging system. This creates detailed logs to enable better supporting. My focus was on testing it. The objective of this is to add entries in the log. As you might surmise, the functionality is not very complex and therefore functional test

cases are indeed easy to generate. Therefore the functional test cases were kinda simple... Hmm, this does seem not right.

I proceeded to question beyond the typical behaviour of the user story- why are we implementing this, who is going to benefit from this, what they might expect from this and so on. The answers that I got via probing were interesting. The intent stated as the prime reason for this user story was to 'enable better supportability by giving detailed information in the detailed logs'. Yeah, seems the typical reason.

On questioning on how it may look in real life, I discovered that this log file could be a pretty long (a few thousand lines) and not exactly machine analysable. Ouch! this means that the poor support guy would be glued to the monitor in a kinda "edit-search mode" looking for potentially interesting information. Hmm.. an onerous task that will consume a non-trivial effort/time to diagnose the problem.

On laying out this potential situation after interrogating the user story team, the team understood that this requires serious rework as the "usability" of this is deeply flawed. The supportability is not getting any better. That is when a light bulb started to glow in me - I found an interesting bug, not in the code yet(as this is yet to be coded) but in the design itself.

Situation #2

The user story in this case was "checkpointing", a set of APIs that allows an developer to implement "application level transactions" that are needed in the system. This enables take a checkpoint i.e. snapshot of the system to be taken before updating the system

with new assets. Post deployment of the assets, in case of any issue, the system can be rolled back to the prior checkpoint.

Similar to Situation #1 the functional behaviour did not seem complex and therefore the functional test cases were simple. Again my nose wrinkled in suspicion, as the test cases designed were too simple. I embarked on the detailed probe and discovered "a critical situation" where the prior checkpoint would be deleted before the current one is completed resulting in a unrecoverable system. The light bulb in me glowed brightly, a serious flaw uncovered, once again not in the code but the "would-be" code. This happened when we dug into the design of the code, assumptions made (note that we were looking for bug related to environment) and questioning led to this potential flawed situation.

The discovery...
An user story is like a "sutra" - an aphorism, that needs to be delved into detail to understand its entirety. And this is needed if you want to test well. Questioning is a key activity to dig into the details as the typical documentation of user story is condensed. Most often the functional complexity of an user story is low, the challenge is in understanding the behaviour of interactions with other stories, environment and the non-functional aspects.

What I discovered is that the act of breaking the "big" into "small" (user stories) makes one forget about who the end user is and what they value. Hence it is necessary to think from the end user's perspective as what they do, how the user story fits in the end user flow and how non-functional attributes of the larger flow matter to the user story. Well the user story should be focussed on the user!

"Sutras" are powerful, as they communicate deep stuff in a few words. To understand the deep stuff, intense questioning is key. Therefore in the Agile context, testing is therefore not anymore an act of evaluation post coding, it is about intense questioning to "sensitise and & prevent defects" rather than "design & execute test cases".

Write less. Communicate more.
Think deeply and may the light flow into you.

Have a great day.

Do less work – 'Test case immunity' can help

June 2012

It is not just seeing what is present.
It is about seeing what is absent.

Test cases and the defects they uncover are central to what we do. We focus on uncovering more with the fervent hope of finding less later. We are also constantly challenged in doing this with less effort and time continually.

In numerous interactions with test and management folks, the conversation always veers to two key questions - (1) Am I doing enough? Is my 'coverage' good enough? (2) How do I optimize and do it faster? I have noticed that often test automation is touted as the solution for (1) & (2) i.e the ability to cover more area with less effort/cost using technology. I understand this, but feel that this does not go far enough to achieve the complete solution for (1) & (2).

This is when I started thinking deeply on the story "The pesticide paradox", particularly for (2). My line of thinking was "How can I ascertain that my software has become immune to some of the test cases and therefore not execute them?" It is not doing faster and cheaper, but it is really about 'not-doing'.

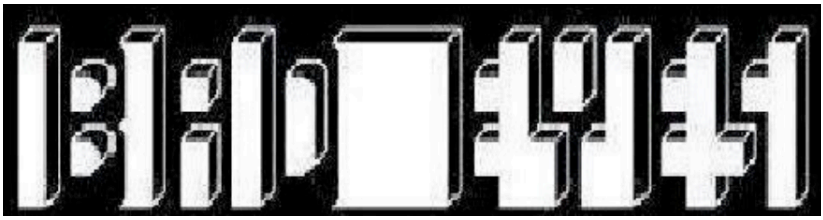
The story of pesticide paradox in brief...
"A poor farmer loses his crop and is advised to use pesticide. The next season around, he sprays the crop with DDT killing the pests and improving the yield. A few seasons later, the pests become resistant to DDT and now he is advised to switch to a different

pesticide 'Malathion'. The yield improves but the story repeats again after a few seasons. This is the story of pesticide paradox, wonderfully illustrated by Dr Boris Bezier in his classic book "Software Testing Techniques".

The pesticide paradox is "The pest that you kill with a pesticide makes the pest resistant to that pesticide". This is used to illustrate the fact, that over time software too becomes 'resistant' to test cases I.e test cases do not yield bugs.

Let's shift gears now... Let's look at defects and what we do with them. We use the defect data and produce reports that provide information about software quality and also about test quality. This is done by examining data related to defect rates, defect densities, defect distribution etc. I.e we pay significant attention to defects. We know that as time progresses,the same test cases do not yield defects, then what do we analyze? Hmm..

Look at the interesting picture below. What do you see? It depends on what you want to see, and at possibly on what distance you see it from. Focus on the black and move the eye farther from the picture and voila, you see a meaningful phrase instead of a mix of fat lines.



What am I getting to? If you chose to see defect information only, analyze them and use the information to make choices, then you are limited. On the contrary if you see "no-defect" (i.e absence of defect) and at the same time shifting to a higher level view of seeing 'defect types'(rather the raw defects), you see new information suddenly, and this will help you find better answers for (2).

Setting up defect types (termed as Potential Defect Types- PDT in Hypothesis Based Testing 'HBT') and then categorizing the defects found into these types, and more importantly analyzing those defect types that have have not surfaced (i.e. no defects of these types) allows us to understand as to which test cases have not still yielded defects..

If it can be proven that the test cases are indeed complete/ adequate (in HBT this is done by assessing two properties of test cases - countability and fault traceability) then the absence of certain defect types indicates "test case immunity" indicative of "the hardening of software". This means that the area of the software being irritated by the test cases has hardened i.e become immune and is clean. Hence focusing on this area of the software is therefore not logically useful and hence these test cases can be "parked". The net result is that we do less work and therefore achieve a higher degree of optimization.

I can visualize you shaking your head in disagreement and commenting "How can I 'park' these test cases not knowing if I may be leaving one 'minesweeper bomb' inside?"
OR
I do this anyway, my experience enables me to figure out the smallest subset of test cases that I need to execute.

My take on this is - Let us do this logically by examining the "categories of empty space" (i.e absent defect types). That is, assess the parts of software system that have become immune to those types of defects that matter.

Remember that we are not examining the actual defects, rather we are examining the test cases that have passed, across the last few cycles of testing, with a clear knowledge of the type of the defect each test case is targeting on. This is examining the 'empty space'. It is however very necessary to ensure that the adequacy of test cases be logically proven, before performing "test immunity analysis".

On a different note, we now know that empty space consists of dark matter that cannot be seen but probably shapes our universe. See the unseen. Enough philosophy.

Mull over this. Every time a test case passes, don't pass over it, use this knowledge of "no defect" to logically analyze "immunity".

Do less work. May the force be with you.
Until next time CIAO.

Purity. Quality. Cleanliness Criteria.

May 2012

Quality is about meeting expectations, properties of a system.
And testing is about assessing how well these properties have been met.

What do we do when we to want to assess the purity of a material? We clarify what the expected purity is, by identifying properties it should satisfy. Then we check if the given material does indeed satisfy these properties.

Properties? Do you remember studying about physical and chemical properties in your school days? Physical property is one that does not alter the material while chemical property is the one that does alter it. Some examples of physical properties are density, viscosity, malleability, conductivity while flammability, enthalpy (of formation), toxicity are examples of chemical properties.

Interesting observation - To assess 'Purity' of a material, we have to check the various '...ity'(s). Purity is really a degree of how well properties have been met. And properties are those that can be observed when evaluating the behavior of the material.

Let us shift gears to software now. One of the key objectives during software development is to assess 'Quality'. Quality is about how well a system/software meets the expectations, similar to the notion of purity. It is no coincidence that they sound similar too. So by extending the previous logic, if 'Purity' is about satisfying properties i.e collection of '...ity'(s), then (software) 'Quality' should also be about satisfying properties too.

Another interesting observation- Quality is also a collection of '...ity'(s): Functionality, Security, Reliability, Usability etc. Cute eh? Let us extend the logic now. The 'ity(s)' of software are really properties of software that we need to check for. And the act of evaluating software quality is really about checking how well the properties have been met. Akin to the categorization of material world properties into physical and chemical, in the software world, we categorize these into functional and non-functional. The former is about the correctness of transformation (of input to output) while the latter is about attributes during the process of transformation.

So where am I leading you to? To a simple understanding that the act of testing is about clarifying/setting-up the expectations by identifying properties and their intended value and then assessing them. Interesting questions emanate when you try to identify the properties and their intended value. And then devising appropriate tests to assess the properties. ISO 9126 is an useful guideline that lists these properties.

This leads us to a scientific approach to understanding expectations rather that rely only on past experience or resort to guess work or gut feel. An approach where understanding of expectations of entity under test is about identifying the properties that matter and then their intended values.

This is termed as a 'Cleanliness criteria' in Hypothesis Based Testing (HBT)- A set of objective (I.e testable) criteria that a system should satisfy. So the next time you test software, identify the cleanliness criteria first I.e list of properties that it should satisfy and the test purposefully.

It is great fun as it throws up more questions than answers. Enjoy!

Properties of a good test scenario/case

April 2012

Trust and proof are on diametrically opposite poles.
Trust you can do well, Prove that you have done well.

We all design scenarios/cases to test a system. How do we know that they are good What are properties that a good set of scenarios/cases should possess?

The prime objective of test scenarios/cases is to uncover potential issues in the entity under test. In the process of doing so, we would like to ascertain correctness.

Let us jump right in the properties that a good test scenario/case should have.

1. A given scenario and associated test cases should be clear on what it is validating i.e. what entity it is testing. This in common parlance is understood as "Requirements traceability".

2. It should be clear as what type of defect it has the power to uncover. This in HBT (Hypothesis Based Testing)is called "Fault traceability". This makes the scenarios/cases purposeful.

3. That the number of scenarios/cases shall be proven to complete. This is a "controversial" statement. In HBT, this property is called as "Countability" i.e. that the number of scenarios/cases are no more or no less. This can be arrived only if the intended behaviour is modelled and then scenarios generated. The number of scenarios can be arrived by combining

the conditions logically and the corresponding test cases generated by combining the data values corresponding to the conditions.

4. That the test scenarios/cases should be staged into quality levels rather than being one to uncover a variety of defects so that these are small and purposeful.

5. That in addition to staging by quality levels, it is good to group scenarios/cases by test types to enable better clarity and purposefulness.

6. The number of scenarios/cases as we progress from the lowest quality level to the highest is shaped like a frustum of a pyramid. i.e. number of test scenarios/cases at the earliest quality level is much higher compared to those at higher levels.

7. That the distribution of positive and negative test scenarios/ cases is indeed good enough. Hmm – how do we know this? Extending point (3), negative scenarios are generated when conditions that are violated are combined while negative test cases are generated when values of data that do not satisfy the conditions are combined. Given this more conditions implies more scenarios (and also negative scenarios) while more data values implies more test cases. As one proceeds from lowest to highest quality level the distribution of -ve:-+ve skews on lower side. i.e. at higher levels, the scenarios/cases are more conformance oriented

8. That the scenarios/cases are ranked in terms of priority of the entities they test and the types of defects that they defect, to

enable intelligent choosing of which to execute when faced with crunch time.

9. That the scenarios when automated, run unattended and as failure of one does not not stop the execution run. It would therefore be useful to design the execution order of scenarios. i.e which scenario to execute in case the current one fails. This can be factored into automation to allow for "long runs" intelligently.

Understanding properties can enable us to assess the efficacy of scenarios/cases and also yield higher efficiencies. The test case architecture of HBT ("Hypothesis Based Testing") captures these nine properties as a beautiful "9-layered shell". The form and structure of test cases in HBT enables better clarity enhancing the effectiveness & efficiency.

Have a great day!

The promising athlete

March 2012

Run. Run fast. Run long. Run well.
Eat. Eat enough. Eat right.

"What is the objective of load test?" I asked the participants attending my workshop.
"It is to find out if the system can handle the load" replied one of them. A circular answer, this does not help!

Another person chimed in "Having multiple users use the system at the same time and check if the system functions properly".

"Does that mean that load test is only applicable for multi-user systems" I asked. Silence followed.

I have observed that the clarity of understanding of some of the common non-functional tests like load, stress, performance, scalability, volume is poor. Possibly because the tests are inter-related. Possibly because the participants did not go beyond of these jargons!

Using "Joe, a promising athlete" as an example I have found it easy to explain the objectives of these tests and improve the clarity of what they are. Let me share this with you now.

Joe is a young athlete, his coach seeing in him a future world champion. Every day Joe spends significant time at the gym building muscles/strength. He lifts weights and is comfortable lifting up to 40kg snatch a few times.

The coach observing that Joe is comfortable with 40 kgs hands him 50 kg weights. Joe is successful, but finds it progressively difficult. After a few minutes he pauses to catch his breath. The coach hands him 60 kg!

"Coach" Joe whines. "Go on Son" says the coach in a baritone voice.
Joe does succeed, his body glistening with beads of perspiration.

After stating this short snippet, I asked the participants in the workshop "So why is Joe whining?"
"Well he is tired" says a participant.
"Why is he tired? I ask.
"It is pretty obvious, he is run out of energy", says another.
I continue with my next question "So why has he run out energy?" knowing fully well that my audience will think that I am pretty dumb.
"Well he has done a lot of work and that is the reason Joe has run out of energy".
"Great. So Joe has expended energy doing work. When his energy has run out, he is unable to continue further" I summarized.
"Now, can you tell me how his story relates to load/stress test?" I ask.

I see the participants eyes brighten and one of them chips in "Oh Yes, Load is about subjecting the system to do "work" using the energy i.e system resources. 60 kg is the maximum Joe could lift as he was stressed out, and therefore stress test is finding out the maximum load that the system can "lift" before the energy ran out i.e. resources are depleted".

“Wonderful guys. Let us summarize. Load test is about understanding if the system can do the typical real life work with the given resources while Stress test is about understanding the maximum amount of work that can be done by consuming ‘all’ the resources”.

Now the coach noticing that Joe is tired gives him an energy drink. After a few minutes Joe is ready to go and the coach hands him 75 kg. Joe now energized, lifts it with aplomb!
“Now what can we learn from this?” I ask.

The participants think for a while and then “Well I think Joe got an additional shot of energy from the drink and he was able to lift more” said the the guy in the last row.
“Wonderful correlation. Yes, he could do more work with the additional energy. Now can you connect this to scalability test?”

After a momentary pause the person on the far left says “Yes, once stressed, additional energy injected allows us to do continue to do more work. Hence scalability test is about checking if the system when stressed out can indeed do more work if given additional resources.”
“Brilliant. Now would one of you summarize this please” I ask.

The short guy with glasses in the first row says “Resources (i.e. energy) enables us to do the various System Operations (i.e. work). The three tests Load, Stress & Scalability” are about connecting these in different ways. I will illustrate this on the board”.

He walks to the board and writes:
Energy (Resources) ---enables--> Work to be done (#System Operations)
Given a set of resources, Can I do the typical #Operations ? ==> Load test
If I use all given resources, What is max #operations that I can do? ==> Stress test
If given additional resources , Can I do more #operations ? ==> Scalability test
Operations = Set of features used to accomplish a job.
And a typical day requires different jobs to be done by different end users of a system.

"Great summary. Indeed simple. Now how does this relate to performance test?"
"That is easy. Performance is do with time, the time taken to the work (operation). Performance test is about measuring if the time by an operation is indeed acceptable (with the given resources)"

"Does this mean that the load and performance test can be done at the same time? The objective however of these two tests are indeed different, is it not?" asks the back bencher.
"Yes, you are right, they may be done together, however their objectives are different" I say.

"Before we conclude, one last question - What is volume test?" I ask, walking to the end of the room.

After a few seconds, a quiet but firm voice behind me says "System Operations is really about processing data and volume test is about checking if the system can indeed process large amounts of data. By the way in some systems, a large volume of data can stress and choke the system".

"Guys, guess Joe and his coach helped us understand Load, Stress, Scalability, Performance and Volume test. I hope you understand that any kind of system (single/multi user) can be subjected to these tests. Real life load is about subjecting the system to various types of operations (concurrently if multi-user) and not just subjecting the system to a blast of a specific operation" concluding the class.

"Yes" chimed in the participants with a big smile.

Thank you guys. I am done with the class.
It is tea time now!
Enjoy.

Do you know the "potency" of your test cases?

February 2012

When you are potent, the world is at your feet.
With time, everything becomes impotent.

We all know that good test cases is the key to effective testing. So what is "good"? And how do we know if test cases are indeed "good"? When I ask these questions to test professionals, they say that a deep knowledge of application domain is required to answer this question.

Let us examine a non-software problem and seek inspiration to answer this question in a scientific manner.

How do we know if a drug that we take to cure a disease is indeed good? If the drug targets a specific bug, targeting a specific area without affecting others, and does its job as quickly as possible to cure the disease, then we can state that the drug is effective. We say that the drug is potent. Do appreciate that the drug potency can be markedly reduced if I develop a resistance to it.

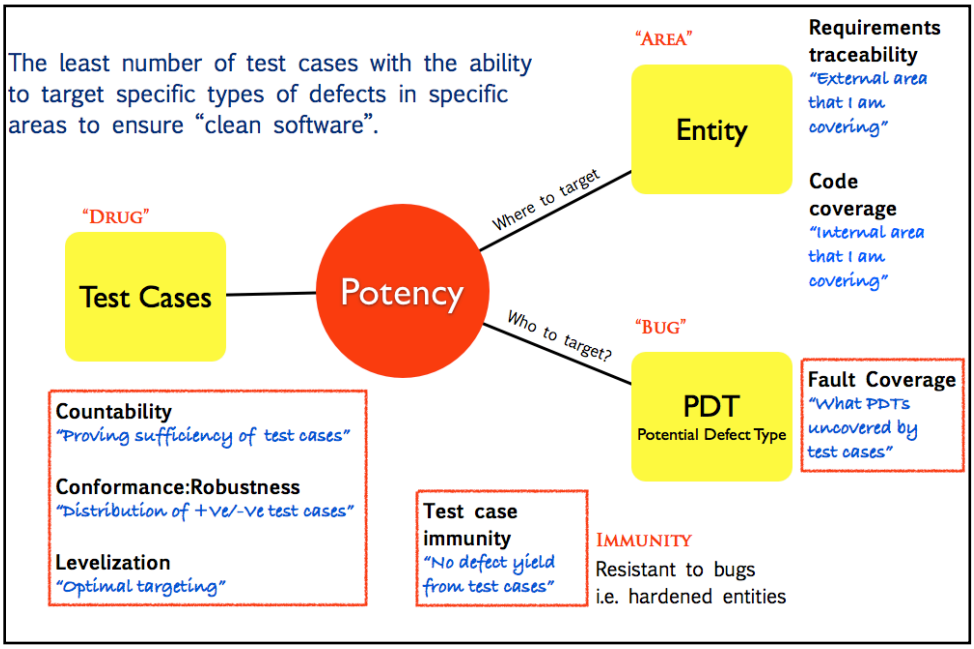
Note the similarity - test cases equated to the drug, potential types of defects to the bug and requirement/feature/code module to the affected area. So how can we use this example to assess the "potency of test cases"? A definition of "potency" is "the strength of the drug, as measured by the amount needed to produce a certain response". Potency is associated with efficacy of an entity, the ability or capacity to achieve or bring about a particular result.

In the context of testing, potency could be defined as - “the least number of test cases with the ability to target the specific types of defects in specific areas to ensure clean software”.

We could assess potency by examining four aspects:

- Strength of drug (aka test case)
- Target area
- Target bug i.e. potential types of defects and
- Immunity

Examining properties of these aspects in a scientific manner could help us assess potency.



Let us dive in. If we can prove that the test cases are adequate, that it covers the area under focus and targets those types of defects that matter then it is a great start. Then examine the notion of "immunity" i.e. Could the system under test be resistant to the test cases?".

Let us examine the key properties/measures of these aspects (of potency) in detail.

1. Test cases ("Drug")

Are the test cases sufficient? Assess the property of countability. What is this? If the behavior of an element under test is described as a set of conditions (i.e. a behavioral model) then an optimal combination of these conditions would result in "countable scenarios" i.e irrespective of who designs, the number of scenarios is indeed the same. Instantiating each scenario with optimal combinations of inputs would result in test cases. The key element to note is that the number of test cases can be logically assessed for sufficiency.

The second measure is the distribution of test cases for conformance/robustness (+/-). This is a function of #inputs and values for each input. Note that inputs can be discerned from the conditions in the behavioral model and the values for each input mechanically generated based on the input specification. Hence this measure can allow us to ascertain if our distribution is indeed good enough.

Finally segregation of test cases into quality levels (See the earlier article "Form & Structure matters in Nov 2011") enables test cases to be purposeful and enables a clear targeting of what type of

defect a test case is targeting (i.e. enable the drug to reach the target area quickly).

2. Entity ("Area")

This is about examining if we are indeed covering the entire area well enough. Measuring how much external area has been covered is "Requirements coverage/traceability" whilst measuring how much internal area has been covered is "Code coverage". Note that these are the typical measures that we use.

These measures focus on the area that we cover, but not the types of defects we are looking for in that area. This is what we discuss next.

3. Potential Defect Type ("Fault coverage")

Tracing the test cases to the potential types of defects that they can uncover enables the test cases to be purposeful i.e. what issues am I interested in uncovering? (This is the central theme of HBT - "Hypothesis Based Testing").

So examining the properties/measures of Countability, Conformance-Robustness distribution, distribution of test cases by quality levels (Quality levels outlined in HBT cookbook http://slidesha.re/qBMNiy) and Requirements/Code/Fault coverage can allow a scientific assessment of potency of test cases.

But could the system under test be immune?

4. Immunity

The pesticide paradox ("The Art of Software Testing by Boris Bezier) states that pesticide that is used to kill a pest makes the pest resistant to the pesticide necessitating newer pesticides to be

created and applied. In the context of software, this could be understood as “software has hardened” i.e. these types of bugs have been removed and probably not present. Hence test cases that are focused on uncovering these kinds of defects may not find any defects.

Examining the yield of test case over time will give an indication of the “resistance or immunity”. Hence tracking the outcome of each test case “defect yield” (i.e. did they result in a defect) every time we run it is key to understanding this property.

The next time you design/review test cases, look at these properties to assess potency.
You want to say “I M POTENT” not “IMPOTENT”.

Cheers.

Single entry, Single exit – Singular purpose

January 2012

Tame complexity by breaking it down.
If you cannot chew, prepare to be eaten.

Functional automation is seen as a key enabler for good testing, especially at a product/application level. Significant money, time, effort are spent to ensure we can do faster, cheaper and better by leveraging technology to do more. This is usually about choosing tools, devising architecture/framework and then cranking out automation code.

And then we discover the ice-berg; that we have to change code to be sync with evolving software i.e. we need non-trivial investment to modify automated scripts. And ouch! it is a pain. Why does it become a pain? Because the script is complex. And what makes it complex? The answer possibly is due to the variety of conditions in the script to check the various behaviors.

As a consultant, when I examine existing automation scripts, I am always amazed at how much some of the automation code tries to accomplish. In a recent engagement while examining the automation code of a product with rich browser-based GUI, I discovered that the automated script checks for data boundaries, default values of inputs, control enabling/disabling, control properties, in addition to the correctness of functional behavior. I am sure that you can visualize the various conditions to check for and therefore the associated long code that is rich in potentially uncovering a variety of issues. The developer of the script is proud

of how much he/she has packed into the script whilst I am smiling and cringing inwardly. The "richness" is the problem!

We as human beings have an amazing ability of dealing with very many things at the same time. Replicating this in a machine is not easy. It is necessary to break a complex activity into multiple simpler activities so that a "machine/automaton can chew this". It is necessary to assess if an entity is in a state that it is fit enough to be delegated to a "machine or automaton". What does this involve? The act of simplification of the activity list by making activities "single minded" to reduce complexity.

So where are we going with this? Reduce the complexity of functional scenarios that are being automated by decomposing this into smaller single-minded scenarios. Taking the above browser-based product as an example, we can decompose into scripts:

1. That check data boundaries
2. That check the control properties
3. That check functional behavior and so on.

What are we doing? Being single minded i.e (a) Writing scripts that are focused on uncovering one (or very few) types of potential defects and (b) Ensuring that each scenario has only one kind of behavior i.e one type of output.

Voila, this means that script has minimal conditions in the code and hence it is more or less straight line code. Wait a minute... Does this mean that the Cyclomatic complexity of the functional automation code is very low? And therefore easier to change/ maintain? YES, you got it.

Good development is about writing less code and less complex code. Note that automation is development, hence good principles of development apply here too.

Single-entry, Single exit code i.e. code with ideally no conditions is the least complex code. Code with least complexity is less prone to bugs and therefore easily maintainable. A good developer lowers complexity by re-factoring code. The same thing an automation developer should adopt.

So as a script developer, what do I have to do? Assess the "fitness of automation" of scenarios that you have to automate. That is, simplify a scenario if it is attempting to do many things. Simplify it by decomposing it into various levels with each level focusing on a specific type of defect. For example a Level -1 script could focus on only checking on input correctness, while Level-2 script could check for interface correctness and Level-4 for functional correctness. In HBT (Hypothesis Based Testing), this is called "Fitness for Automation" with scenarios being "level-ized". This helps in making the scenario "chewable" by the "automation".

We now have scenarios that are single minded (i.e. focused on finding a potential defect type), a scenario defined and understood as a singular behavior resulting in an output. When we convert these into automated scripts, we have a "single entry-single exit" script. A script with least set of conditions (ideally zero) and therefore a "singular purpose".

It is often assumed that a good framework is a good enough for building flexible scripts. This is true, but represents only half the story. The other important half is the structure of a scenario and its "fitness for automation". A good structure wherein a scenario

has a “singular purpose” enables the script to be “single entry-single exit code” i.e. least complex code.

As we saw in the Dec 2011 column(‘Taming complexity’) complexity is an enemy and friend. An enemy because it can faze you, freeze your thoughts and numb you. A friend because it depicts richness, the ability to do very many things. The trick is to be rich (level-ized) but not be fazed by it (singular).

A clear goal focus sharpens intellect. A scenario with a clear goal results in a “singular purpose” script that satisfies of the key attribute of “single entry-single exit” to enable you evolve with least pain.

So the next time you convert a scenario into an automated script, make the scenario “chewable” or be prepared to be “eaten”. Have a great 2012.

Taming complexity

Dec 2011

Everything is complex.
The trick is to make it look simple.

In my meetings with senior management and technical folks in high technology companies, the one phrase that I frequently hear is "My system is complex". When anyone utters this phrase I smile inwardly. The way I interpret this is (a) you do not understand the system or (b) you are trying to impress me.

Complexity is an enemy and friend. An enemy because it can faze you, freeze your thoughts and numb you. A friend because it depicts richness, the ability to do very many things. The trick is to be rich (in features/functionality) but not be fazed by it. Philosophical eh?

We all have intuitive understanding that complex systems demand 'more' or thorough testing. More testing is not about doing it frequently, it is really breaking down the complexity and and ensuring that these aspects are well covered. We know that with experience, we handle this well as we are good at decomposing the problem and solving it.

Let's probe as to how we possibly do this. Let"s start from the beginning. "What does it mean when we say "My system is complex"?. It simply means that there are numerous possible behaviors. Depending on the value(s) of the inputs, the behaviors invoked are different and therefore generates different results. So one of the aspects that make a system complex are the various

conditions that govern the behavior. So if the entity''s business logic i.e. behavior has quite a few conditions, then there are very many paths (I.e scenarios) to test. To do this well, it is necessary to model the behavior using a suitable technique so that the complexity of the numerous conditions do not faze us. This is "Behavioral complexity".

The next aspect of complexity is "how good is the behavior". What does this mean? It is about the expectation of certain attributes that the end user may be expecting when displaying the behavior, like speed/performance, security etc. In some cases the expectations of the attributes may be very high and therefore requires a good architecture/code to satisfy the demanding attributes. This is "Attribute complexity".

Let's move inwards to see the role of the internal structure towards complexity. The number of elements (i.e. moving parts), that make up the system, their interconnections and the way they are interconnected is what contributes to "structural complexity". Structure at the highest level is the deployment architecture, while it is code structure at the lowest level and the system architecture at the middle level. More moving parts, dependencies, interconnections at any of these levels implies the system is "structurally complex".

Structure is not simply about code aspects, it is also about the structure of data. In some systems, the underlying data models can be involved and interesting and therefore we have "Data complexity".

The final aspect of complexity emanates from the environment- the number of environments that the system has to support, the

dependencies the system has on the environment (e.g. setup, configuration...) and the rules of the environment that the system has to comply with. This is Environment complexity".

Let us a do quick recap now, we have broken down complexity of a system into five aspects:

1. Behavioral complexity - the number of conditions that govern the system behavior
2. Attribute complexity - "how good is the behavior"
3. Structural complexity- #parts in the system, #interconnections and the method of interconnections.
4. Environment complexity- environments, configurations, rules of the environment
5. Data complexity- volume/rate of data, inter-relations between the data (data model).

So when somebody says their system system is complex, decompose it into these five aspects and understand and rank these. This will allow you to get a handle on complexity and stay on top it, not be crushed by it!

Good testing is about understand the “complexity contributors”. apply suitable techniques, expending the necessary effort and adopt a risk-driven approach to validation to maximize business results. Hypothesis Based Testing (HBT), decomposes the system into five elemental aspects (Data, Behavior, Structure, Environment, Usage) and one of the one core concepts “Complexity analysis” is useful in taming complexity. Lookup http://slidesha.re/qBMNiy if you are curious about HBT.

So the next time you tell someone “my system is complex” Think! If you understand your system well, then you will indeed

decompose it and describe it in simple terms. Albert Einstein said "If you can't explain something to a six-year old, you probably don't understand it". Hmm. Imagine explaining General Theory of Relativity to a six year old!

Systems or life are complex i.e rich. Simplify it and you see beauty. The utter simplicity that hides the raw brutal complexity, is what makes an end-user fall in love with the system. Well I am an Apple fan!

As a tester I enjoy grappling complexity because I am confident of taming the beast.

The second law of thermodynamics (Entropy) is alive and kicking in software systems too! Enjoy the richness of complexity and the beauty in making it simple.

Let your new year be filled with richness laced with the the beauty of simplicity. Happy New Year.
CIAO.

Form and structure matters

November 2011

Goodness is holistic.
Beautiful outside. Strong inside.

Riding a roller coaster is great fun. Irrespective of age the ride is filled with thrills. The rush you get from roller coaster is indeed out of the world. The slow climb and then a sharp fall with gut wrenching twists and turns is an exhilarating experience.

We look at this engineering marvel with AWE and RESPECT. How is it possible that this steel lattice manages to give so many thrills and still hold a safety record, better than a toaster?

The answer is simple–it's the form and structure of the roller coaster. The physics behind the form and structure are key to the forces that we experience and the thrills we enjoy.

"Form and Structure" is the integration of design and engineering. It is ubiquitous, from nature's leaves and honeycombs, to man-made structures. Form is about the external shape, the way it is presented, while structure is about the way the elements are arranged/composed. Both are critical to any design activity and this applies to test design too.

Typically our belief is that effective testing is the result of good test cases. The "goodness" of test cases is normally associated with the test case contents. Are good test cases the result of one's experience or the strength of the techniques only?

The architecture consisting of external form and the internal structure play a vital role in ensuring that the test cases are adequate yet optimal and can be generated and automated rapidly. The form gives the shape to test cases allowing us to see a variety of dimensions related to adequacy, review-ability, automate-ability, and product health. Structure is what allows assembling the various elements of a test case to create the shape that allow seeing the various dimensions.

Hypothesis Based Testing (HBT), a personal scientific test methodology pays significant attention to the form and structure of test cases. The test cases in HBT has a nine-dimensional form (or shape), with each dimension focusing on a specific attribute of goodness of the test case. The nine dimensions are:

1. Cleanliness level - What level of quality or cleanliness are these (test cases) focused on?
2. Entity being validated- Is it an elemental component, technical feature or business flow?
3. Potential defect type expected to be uncovered - What type (or class of defect) is it expected to uncover? And the types of tests needed to uncover these.
4. Focus (Conformance or Robustness) - Does it validate the correct use or recoverability in the case of abuse?
5. Priority/Importance - The perceived importance of the test case to the overall customer experience.
6. Stage it needs to be executed - At what stage of development is it expected to be executed?- Sanity check, Validation check, Regression.
7. Intended execution frequency - How frequently do we expect these to execute? Daily, weekly, once a build, etc.
8. Optimal sequencing or threading of test cases - Which is the next text case to execute when the current one fails? This is

especially useful for test automation to ensure that we do not "baby sit" the scripts to ensure unattended long runs.

9. Mode of execution - How do we intend to execute? Manually or Automated?

The first FOUR dimensions focus on efficacy aspects of test cases while the remaining FIVE dimensions focus on efficiency of execution. The segregation of test cases in the first two dimensions (of cleanliness level and entity) enables rapid automation by ensuring short purposeful scenarios and therefore simple small scripts.

as an entity under test to be validated and consists of short scenarios that are staged by cleanliness levels making their conversion into scripts easier and simple to maintain.

As to the structure, in HBT there is a clear notion of behavioral scenarios for each element under test, with each behavior assessed by a set of stimuli i.e. test cases. In addition to the structural aspect of requirements traceability, there is an interesting notion of FAULT Traceability that associates each scenario with the potential defect that they can uncover. At the lowest level, structure focuses on the simplicity or the terseness of how they can be written, to ensure rapid test case design yet being very effective.

Test cases are not to be understood as a sequence of steps. It is really an optimal combination of inputs to stimulate a behaviour (scenario) thoroughly. Tracing the stimuli to potential types of defects it can uncover, makes the test cases purposeful and effective. Finally, detailing the preconditions and execution steps aids in converting these into automated scripts.

The contents of the structure can be documented for posterity in a form that is terse or detailed; this is probably governed by the organizational process. Form and structure are integral to the goodness and should not be confused with documentation detail. After all a beautiful rose can be described in a single sentence or in an elaborate poetic form.

In summary, the form and structure enables:

- Test cases to be goal focused i.e. what types of defects to uncover
- Clear assessment of effectiveness of test cases
- Enable selection of appropriate test cases to optimize execution
- Enable objective assessment of "system health"
- Smaller scripts aiding rapid automation with low maintenance

So when you design test cases, do not just focus on the contents, see if there is a form and structure that allows us to see the various dimensions of goodness. Ultimately a good form and structure brings about the aesthetic value, help to see and enjoy beauty in what we do, and get a satisfying experience rather than just test!

On a lighter vein, the next time you ride a roller coaster, do not think about testing - ENJOY the ride!

Have a nice day.

The mistaken notions of structural testing

September 2011

Structure is what holds the various parts of the system together.
And a robust structure is key to good functional/non-functional behaviour.

Whenever I conduct a session on testing, I am always amazed at how misunderstood structural testing is. Whenever the topic of white box comes up, the participants, typically full time test professionals involved in testing the whole system, say that this is out testing the code by executing the various statements, branches or paths. The immediate conclusion is that this is to be only done by the developer and therefore does come under their purview. When probed further they say "white box testing" is about doing coverage based testing.

Hmm.. Let us think on this..
Let us commence with a discussion on the phrase “white box testing”. Sadly the phrase as used, is incorrect, it should be "white box test techniques". That is, a set of techniques that use the structural properties of the system to evaluate behaviors. The premise being ‘that the system could fail because of faults in the structural aspects of the system’.

So what are the structural aspects that we would like to examine? Before we commence on this line of reasoning, let us ask ask a basic question as to the meaning of structure. What are the elements that constitute structure? What are properties of the structure that we want to evaluate?

A set of parts and their linkages constitutes the structure. A part in turn is built using some "building materials". Our intention is to ensure that the structural aspects of these parts and their linkages are correct, providing us with a robust scaffolding to deliver functional/non-functional behaviors.

So what does a "part" mean in our context?
The building materials would be the technology/languages at lowest level, components/subsystems at the middle level and other systems/environment at the highest level. Therefore what a part is, is level dependent. This could be API/object at the lowest level, it could be subsystem/tasks/services at middle level and other systems to interconnect with at the highest level.

What about "linkages"?
Only when all the parts are "connected", do we get the whole system. The linkage is done in two ways - calling & sending i.e. calling a API/service and sending data. Think about this - when you need help from a friend, you can call on the phone or send a text message. So the linkage could be a function/method call, sending data via messages or sharing the data via global/ environment variables/files/databases.

So let us connect all these. We want to ensure that the materials that make up the parts have been used correctly and that the parts have been linked correctly. At the lowest level, the part is constructed using flows that are sequential/recursive/parallel and using materials from the environment i.e. resources. These parts are then linked with parts by interfaces that are call/data based. But it is important to keep in mind as to what "level" the part is - lowest level code structure, system architecture at the middle level and deployment architecture at the highest level.

Employing this line reasoning, don't you see that as test professionals it is our bounden duty to ensure that the structure is indeed robust and that we have to validate correctness of the structure at three levels:

1. Understanding the deployment architecture and looking for issues in at the higher order structure. Do we know dependencies on other systems, data formats of linkages, latencies, environment resource availability, the plethora of versions of the environment etc.
2. Understanding the system architecture in terms of flows of control/data, sequences of usage, error possibilities and exception handling keeping in mind the non-functional aspects like performance, security attributes too.
3. And at the lowest level, understanding the code structure in terms of control and data flows and resource usage. This may be difficult to do for a full time test professional as this may require a higher level of detail of code structure and hence may be better done by a developer. Nonetheless what needs to be validated can be hypothesized by the tester and the act of execution delegated to the developer.

Let us discuss two scenarios to illustrate this..

Scenario #1: Joe goes to the doctor after a rough night and states that he was feeling very uneasy, with a mild pain the chest, had excess sweat and felt pretty tired. The doctor after external examination comes to a conclusion that he needs to be admitted for a by-pass surgery. I can hear you say "No way!".

Scenario #2: Joe goes to the doctor after a rough night and states that he is unable to breath normally when sleeping. He says that he breathes though mouth and gets up in the morning with a

sore throat. The doctor says that he needs to a detailed examination by cutting open the nose in the middle to uncover the problem. Ouch!

In scenario #1, the act of probing is done via external examination whereas in scenario #2, the examination in intended to be internal! I am sure you would wince at both these scenarios. You would probably expected a meaningful and judicious combination of external and internal examination.

That is what is expected of us too! So never think in terms of pure black or white, it is always black and white. Look at the structure from three levels and when you require a detailed examination of the lower level structures, work with the developer. And for heaven's sake do not use the term "White box testing".

Remember the structure is very important as it provides the scaffolding for the system and needs to be evaluated. A poor structure has a significant bearing on the correctness of attributes like performance, load handling, scalability, reliability, security etc. Note functional correctness may be still be delivered with a poor structure, it is a just a matter of time before the cracks show up!

I am sure you will agree with me that the probability of faults in a system is directly proportional to the complexity of the system. Some systems are functionally complex (i.e. number of conditions) whilst some systems are structurally complex and some systems are attribute-wise complex(i.e. need to deliver a simple functionality but may need to support of millions of instances). Understand that structure plays a vital role in the last two.

In HBT (Hypothesis Based Testing) the three core concepts of Quality Growth Principle, Complexity Analysis and Techniques Landscape enable one to scientifically understand the complexity, when and how to evaluate the structure.

Remember, you are also responsible for structural aspects of the system. If the building falls, you will be famous, but you probably don't want be there on that pedestal!

Cheers!

How many hairs do you have on your head?

August 2011

Do we measure the distance before we back up a car? No!
We approximate the distance and refine continuosly. And it is natural.

One of my favorite questions that I ask when I mentor testers at companies is "How many hairs do you have on your head?". I am always amazed at the answers that I get from the participants.

Some give me a number of 10,000 while some enthusiastic participants give me a number of ONE million. Isn't it amazing that we have of variance of 1000x for such a simple question, the one that we are probably very familiar with. The participant profile typically senior and mid-level engineers. Nonetheless the answers that I get from senior participants are equally crazy. Why is there such a large variation?

My hypothesis is that engineering staff find it difficult to deal with uncertainty, those aspects whose "countability" is fuzzy. So the typical answers seem to be based on seat-of-the-pants approach. In the few cases where we've encountered a similar situation and solved it well, the estimates turn out to be correct, otherwise it's simply a shot in the dark, praying to God that it turns out to be correct.

It is important to understand that an exact value may not be required, rather it is the reasoning that allows us to come up with a value, and constantly improving the reasoning to better the value. The simplest way to answer the "Hair question" is to compute this by multiplying the density of hair and area of head

approximated as square. The answer can be refined by improving the reasoning, that the hair density varies across the head and that we may need to consider the curvature factor of head, hair shed rate etc.

So what are we doing? Rather than just guess somehow, we have tried to construct a simple formula that is continually refined to better the outcome. Note that what we have done is the age old problem solving technique– ”divide and conquer”.

The act of approximation is very natural, it is in fact part of our instinct. Think about this - we do not take measured steps when we walk, we do not calculate the exact distance when reversing the car and so on. Our natural learning system continuously learns in the background and constantly adjusts the variables to refine the approximation.

Let me illustrate how “scientific approximations” is useful in testing... One of the common questions that gets asked frequently is “How much time/effort will it take to test this?”. A correct answer is elusive. A group of people believe that this can be answered only by people with rich experience, whilst some folks in the community believe this question is stupid as this cannot be estimated. However considerable effort has been expended in building empirical models that would magically answer this question, but they still seem to fall short. What is important here is not the exact answer, rather it is the reasoning that enables us to scientifically approximate. Rather than attempting to answer such questions in a simple binary fashion, a reasoning that identifies the variables involved and connecting them by a formula, that is continually refined allows us to find the right answer. Some of the variables may be #scenarios, #cycles, times involved in

understanding, design, automation, execution, mode of execution, volatility of a feature, types of tests etc.

I have observed that when people design load, stress, performance, scalability and volume tests, the test data values used for load profile, size of data are seemingly good guesses. Scientific approximation to estimate the load could be based on these variables- types of end users, #users per user type, features used by an end-user, frequency of their usage, seasonal variations, rate of arrival, growth of business and then connecting these by formula. Continuing on the same train of thought, the data size (volume) can be estimated by identifying the data generating transactions, size of data generated per transaction, the retention period(after computing the number of transactions).

As a professional tester, we encounter a variety of situations that require us to come up with a value/number. Just because we don't see a direct connect to this value does not mean that we resort to "guesswork". What is needed is scientific approximation, the keyword is scientific.

A few years ago I read a wonderful book "titled The Art of Profitability" by Adrian Slywotzky. This book deals with "how to deliver higher profits", where David Zhao an extraordinary teacher beautifully breaks down the problem as a collection of various profitability models and teaches this to his protege Steve Gardner, a CEO of an ailing multi-billion Dollar company. The teacher uses a Socratic style of teaching approach to enable the CEO to discover reasoning. In one of the chapters, when the teacher asks the CEO for profitability percentages for a product line, the CEO does not have the answer handy. This is when the teacher explains the importance of approximations. In fact he asks the CEO "How long

would it take to cart Mount Fuji away with dump trucks, one truckload at a time?" and expects the answer in a minute. This book inspired me to develop the "Approximation Principle" that is a core concept in HBT(Hypothesis Based Testing).

So the next time, when you encounter a curved ball, a question that does not seem to have clear answer in terms of "a value", don't react immediately. Pause, reason, identify the various variables, connect them with a simple formula, compute the first value, test it quickly and refine. Sounds involved and time consuming? Try it once and you will be surprised at how quickly this can be done. I have tried this many times and it has been fun.

Want to try it now? You have been doing a great job of testing applications/products for your company. How much money do you think you have saved for your company because of your good testing? Sounds like a good ploy to demand raise!

Stop counting your hair now and start thinking!
Cheers! Enjoy the fuzziness.

Landscaping – A technique to aid understanding

June 2011

Good questions matter more than the availability of answers.
It is not about what you know, it is about you do not know.

In the last article ("The diagnosis") Joe faced with the problem of understanding a new application in a domain that he is is unfamiliar with, has the "Aha" moment at the doctors's office during the process of 'diagnosis'. He sees parallels in doctor's questioning technique to diagnose the problem and his problem of understanding of the application. He understands that decomposing the problem into information elements and establishing the connections between these elements enables him to come up with good questions to understand the application. Voila!

Joe figured that understanding an application is not just about walking though the various features via the user interface(assuming that the application has an UI), it requires a scientific/systematic walkthrough of various elements commencing from the customer's needs/expectations and then into the application's deployment environment, architecture, features, behaviour and structure.

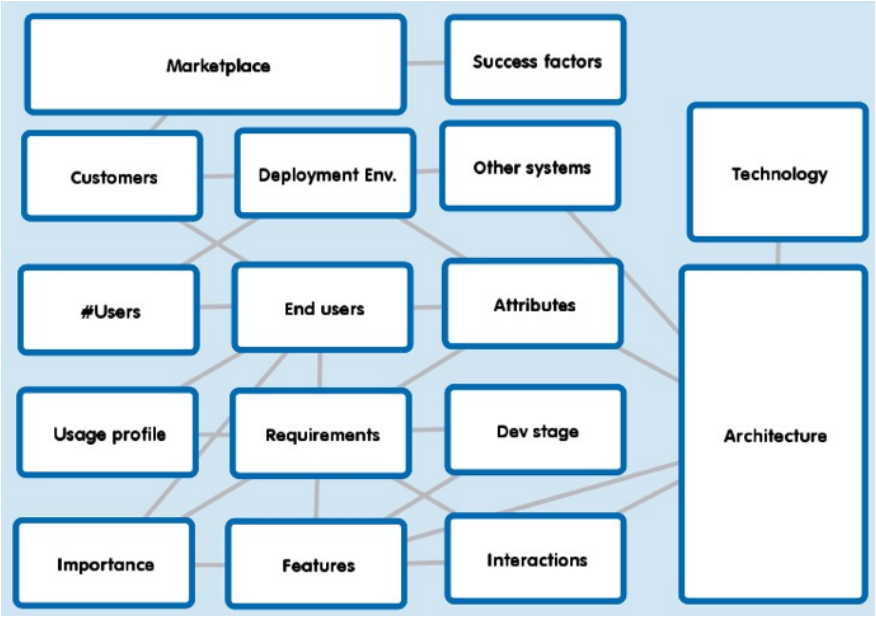
This is a technique called Landscaping, a core concept in HBT (Hypothesis Based Testing) that enables one to systematically come with meaningful questions to understand the end users, application and the context. It is based on the simple principle "Good questions matter more than the answers. Even if questions do not yield answers, it is ok, as it is even more important to know what you do not know.

Now onto Landscaping in detail...
The objective of testing to ensure that the product or application meets the expectations of the various end users. That said, how does one get into the mind of the end-users to be able to appreciate expectations that they have? Let us dissect the problem...

Firstly start from the outside - the marketplace and various types of customers in the marketplace where the application is going to be used. Then identify the various types of end users (or actor) in each type of customer in the marketplace. Now identify the various use cases/requirements for each user type and then the technical features that constitute the business use case or requirement. Now viewing each requirement from the perspective of the end user, identify the attributes that matter for each requirement/feature and then ensure that they are testable.

Then move inwards. Understand the deployment environment, structure of the application (architecture) and the technologies used. Moving further in, identify who uses what feature, the profile of usage, ranking of a feature/requirement in terms of business importance and then the conditions that govern the feature. Finally understand how each feature/requirement affects the other features/requirements and then understand the state of a feature (new, modified, status-quo).

Let's picturise this..



oogWhat are we trying to do? We are trying to create a mind map that consists of various information elements, then question to understand the information elements and more importantly establish the connections between the various elements. The act of understanding the individual elements and their connections results in questions that aid understanding. As we proceed, we establish a clear baseline of end user types, requirements/features, attributes, environment, usage profile...

Let us look at some of the questions that emanate by applying Landscaping...

Marketplace	What marketplace is my system addressing? Why am I building this application? What problem is attempting to solve? What are the success factors?
Customer type	Are there different categories of customers in each marketplace? How do I classify them? How are their needs different/unique?
End user (Actor)	Who are the various types of end users (actors) in each type of customer? What is the typical/max. number of end-users for each type? Note: An end user is not necessarily a physical end user, a better word is 'actor'.
Requirement (Use case)	For each end user, what do they want? What are the business use cases for each type of end user? How important is this to an end user - what is the ranking of a requirement/feature?
Attributes	What attributes are key for a feature/requirement to be successful (for an end user of each type of customer)? How can I quantify the attribute i.e. make it testable?

Feature	What are the (technical) features that make up a requirement (use-case)? What is the ranking of these? What attributes are key for a successful feature implementation? How may a feature/requirement affect other feature(s)/requirement(s)?
Deployment environment	What does the deployment environment/ architecture look like? What are the various HW/SW that make up the environment? Is my application co-located with other applications? What other softwares does my application connect/inter-operate with? What information do I have to migrate from existing system(s)? Volume, Types etc.
Technology	What technologies may/are used in my applications? Languages, components, services...
Architecture	How does the application structure look like? What is the application architecture look like?

Usage profile	Who uses what? How many times does a end user use per unit time? At what rate do they use a feature/requirement? Are there different modes of usage (end of day, end of month) and what is the profile of usage in each of these modes? What the volume of data that the application is subjected to?
Behavior conditions	What are the conditions that given a behavior of a requirement/feature? How is each condition met - what data (& value)drives each condition?

Summarizing Landscaping is a core concept in HBT (Hypothesis Based Testing) that identifies the various information elements and the process of understanding the element details and their connections enables questions to arise. These questions when answered allow one to understand the application(system), customer and the context, and construct a clear baseline for subsequent stages of testing.

We have also seen that when applying this, we uncover the missing parts of the puzzle and this has helped us to improve/fix the requirements. In a nutshell, good questions aid in early defect detection/prevention and we have used this to test requirements and not only code.

Have a great day!

Section 2 – Observe & Learn

This section contains articles that focus on learning to do better via keen observation(s). Going beyond scientific thinking which is is a stepping stone to doing well, to carefully observing the way things are done, learning from these, so that we may get to the next plateau of doing better.

Using mirrors to enhance quality, to change oneself, seeing that a catastrophe is a result of a perturbation of a fault chain, observing doctors, dogs and people in other disciplines resulting in some fascinating learning, are what these articles highlight.

Observe. Reflect. Learn.
May you reach the next level.

Mirror, mirror on the wall, who is the fairest of all?

July 2012

A good mirror reflects what you are.
And what you see changes you.

Snow white and the seven dwarfs is a classic story we all grew up with. The story illustrates very many things, in addition to the power of love.

I have been inspired by the mirror in this story. The mirror that reflects how good I look, every day. As technical folks wanting to deliver the highest quality, we do very many things via better techniques, tools and process. How do we know if these are yielding results? Reflect. Use a mirror. A dashboard containing measurements that matter, that is real time, that helps us adapt constantly and change for the better.

Recently I had interesting conversations with senior QA folks from non-software domains. The gentleman from the fashion and apparel industry said "We are a labour intensive industry, and a lot of variables/factors like fabric, thread,people's mood, equipment etc affect the quality of the apparel we make. And customers don't tolerate poor quality. Know what we do? We implement continuous feedback via a real-time dashboard." Aha - the mirror!

I started thinking on what we do too- we collect measurements related to quality,progress and present them as reports/charts. Most often these are intended for managerial decision making. The person from the high tech manufacturing domain seemed to read my mind, he pitched in and said "It is not about creating great

dashboards to help the manager, it is about empowering the people on the shop floor to assess the situation and make on-the-spot decisions to course correct rapidly. It is about changing the behaviour".

The fashion and apparel gentleman added “It is not just collecting measurements related to defect counts, it is knowing about kinds of defects produced and their distribution over time”. That is when I related to HBT, where the focus is on defect types.

The spectacled gentleman from the high tech automotive industry added "Note that it is great to know about defect types and distributions,but in our industry, we sensitize ourselves deeply to prevent them, as the cost of fix is very high in our industry. We do quite a few things like modeling, simulation to ensure to ensure 'wellness' not 'treat well'".

Hmm...it is not just enough to see the issues in the mirror and work on them, it is not just validation, it is about embedding quality in. A mirror that changes me.

The manufacturing gentleman added "We develop components for high technology industries, the key is to understand the context of usage and look for defects. Testing a component alone is insufficient, we need to visualize the usage context”.

The bulb lit up... The background needs to be reflected too, not just my work. Wow.

The lone gentleman from the software (mobile) commented "We do not always need the "ultimate quality", in certain situations we can tolerate some deficiencies". Not wanting to offend the others he

clarified "This is the concept of 'technical debt'. The tolerance to deficiencies is simply not there in mission critical industries, where product shelf life is long. Whereas in our industry (mobiles - fast moving products) shelf life is low, hence there is tolerance for some deficiencies".

Hmm. This implies that sometimes 'my' reflection is not as perfect, but heck it is fine in certain situations.

After the conversation, I reflected. All we need is just a mirror to do better. Reflect work outcomes, along with the background (usage context) and tolerance of the image for that context! This changes you. It changes your approach/behaviour to quality and therefore how and when you apply.

Find your mirror. Those set of measurements that help you understand the quality of your work and the quality of the product. Those that can change you. Change you do deliver excellence. The perfect reflection.

Mirror, mirror on the wall, who is the fairest of all?
I always want it to me i.e. my product.

Have a great day.

Reflect and change

February-March 2013

It is not about knowing the outside.
It is about changing the inside.

In the article “Mirror, mirror on the wall, who is the fairest of all ?” published in July 2012, the focus was on metrics and how you can use them build a great product. This article continues in a similar similar line, with emphasis of how these can help you drive change in your behaviour. To change behaviour to do better. The prior article was on how reflections assess the product quality, while this is about changing yourself.

I used the wonderful story of Snow White and the Seven Dwarfs in the previous article and am motivated to use it again. In this story, the queen asks the above question everyday to the mirror and feels very happy when the mirror tells her that she is the most beautiful one. And the real story commences when the mirror replies "Snow White is the most beautiful".

A mirror is a wonderful device; it reflects you, it tells you instantly how you look - complexion, girth, emotion, goodness, aberration etc. You can look at the mirror and know what is happening and what action you should take. If your skin looks parched, dry and lifeless, you know that you have to hydrate and nourish it with oil. And that is exactly what metrics should do; to reflect the situation clearly so that we can do something positively. And a talking mirror - Wow!

Measurements that we perform and the metrics we collect should be purposeful, to reflect the status of: progress (of testing), quality (of system), adequacy (of tests) and process (aspects like efficiency, productivity, optimality...).

Just like how the mirror makes us understand the situation and changes how we respond to it by changing our behaviour, good metrics should ultimately result in changing our behaviour.

So what are the behaviours that we would like to change ? Being more effective in uncovering defects, being more efficient in testing, and driving behaviour change to prevent defects.

Knowing the number of test cases and their distributions by features (entities they test), by test types, by quality levels, by positive:negative distributions, reflects if the test cases can be effective. Targeting features that are newer, riskier, complex would imply that the number of test cases should be high in these cases. Distributions by quality levels and test types are called “test depth" and “test breadth” respectively in HBT (Hypothesis Based Testing), and the product of breadth and depth is an indication of test coverage.

To “mirror” efficiency, the typical metric is the percentage of test cases automated, while the interesting one is “immunity” (in HBT), that measures the consecutive pass rate of a test case to ascertain the parts of the system that have become “immune to defects”.

Finally the best behaviour change is when less defects are detected with lot more prevented. The metrics that can give us an indication are the number and kind of the questions that we ask and the

number of potential defect types (PDT in HBT) that are we are looking for.

Capturing metrics that at best are objective indicators of some facets of the system, are not useful as they are mere statements of "what of the system", but do spur us into action. So when we measure a 'Defect severity distribution', what does it mean?

So if you want to "spur into action" and change the behaviour you need the "talking mirror". Take a look at the metrics you collect and ask if they change your behaviour.

"Mirror, mirror on the wall, who is the fairest of all?" And the mirror replies "You!"
Cheers. Have a great day!

Seven consecutive errors = A catastrophe

October 2011

A fault when irritated results in a failure.
A domino of faults when perturbed results in a catastrophe.

"A typical accident takes seven consecutive errors" quoted Malcolm Gladwell in his book "The Outliers". As always Malcolm's books are a fascinating read.

In the chapter on "The theory of plane crashes", he analyses airplane disasters and states that it is a series of small errors that results in a catastrophe. " Plane crashes are much more likely to be a result of an accumulation of minor difficulties and seemingly trivial malfunctions" says Gladwell.

The other example he quotes is the famous accident - "Three Mile Island" (nuclear station disaster in 1979). It came near meltdown, the result of seven consecutive errors:

1. blockage in a giant water filter causes
2. moisture to leak into plant's air system that
3. inadvertently trips two valves which
4. shuts down flow of cold water into generator but
5. backup system's cooling valves are closed - a human mistake
6. and the indicator in the control room showing that they are closed is blocked by a repair tag while
7. another backup system, a relief valve was not working.

This notion is reflected in the book "Ubiquity" by Mark Buchanan too. He states that systems have a natural tendency to organize themselves into what is called the "critical state" in what Buchanan

calls as "knife-edge of stability". When the system reaches the "critical state", all it takes is a small nudge to create a catastrophe.

Now as a person interested in breaking software and uncovering defects, I am curious to understand how I can test better. How do you ensure that potential critical failures lurking in systems that have matured can still be uncovered?

Let us look at what we do- We stimulate the system with inputs (correct & erroneous) so that we can irritate latent faults so that they may propagate resulting in failure. When the system is “young”, the test cases we design are focused on uncovering specific singular faults. i.e a set of inputs that can irritate singular faults and yield possibly critical failures. This is possible because the “young system” is not yet resilient and therefore even a singular fault bumps it up! We then think that our test cases (i.e. combinations of inputs) are powerful/effective. But these test cases do not yield defects later as the system becomes resilient to singular faults.

As the system matures we need to sharpen the test cases to irritate a set of potential faults that are tied to create a domino effect to yield critical failures. Creating test cases to uncover singular faults in a mature system is no more useful. Is is necessary that test cases be at a higher level of system validation (i.e have long flows) and have the power to irritate a set of probable faults.

So can we resort to uncovering critical failures only via testing? By creating test cases at higher levels that have the power to uncover multiple types of faults? Not necessarily. We can apply this thought

process at the earlier of design/code too. Using the notion of sequence of errors and understanding what can happen.

If your drive in India you know what I mean. The potential accident due to a dog chasing a cow, which is charging into the guy driving the motorbike, who is talking on the cell phone, driving on the wrong side of road, encounters a "speed bump", and screech *@^ %. You avoid him if you are a defensive driver. Alas we do not always apply the same defensive logic to other disciplines like software engineering commonly enough.

Have a safe day!

Musings & Inspirations: Learning from non-software disciplines

July 2011

Innovations happen when you look outside your discipline.
Being constantly curious is what makes life enjoyable.

As a moderator of the panel discussion on "Roadmap to higher quality" at SoftTec Asia 2011 at Bangalore, with three panelists from non-software disciplines- Pharma, Food/Beverages and Automotive industry, my job was interesting, of seeing parallelisms and looking for inspirations from other disciplines.

The panelists were from the quality assurance department and they talked about how they assure quality in their disciplines. They talked about standards/regulations, design stage behaviour analysis, statistical sampling and interesting ways to uncover defects.

As a keen observer of their ways of doing and reflecting on our practices, I discovered some interesting parallels and inspirations for software testing. I have captured my musings and inspirations that you should find interesting.

Goal-focused versus activity-based standards

These industries are regulated and the product has to comply with standards before they are released. Hmm... we in software also have standards too. But there seems to be some key differences. The standards that we follow are "prescriptive"; they describe the steps that one has to do. On the other hand, the standards in the above described industries are probably "descriptive".

They state as to what is expected in the product, they set a baseline to comply with. They communicate clearly as to what is expected, in what is termed as 'Cleanliness criteria's in HBT (Hypothesis Based Testing).

The big difference is the whilst we in software are focused on activities and ensuring consistency of these, our friends are focused on the outcome or the goal. BIG DIFFERENCE.

Staining the system to see defects

To uncover defects in software, we design test cases which when executed uncover defects. In the beverage industry, there is an interesting way of uncovering defects. Here used bottles are cleaned and filled with the beverage. So how do we know the bottles are indeed clean? My colleague from the Food & Beverage industry mentioned an approach that I found very interesting.

When bottles are cleaned, the expectation is that moulds shall not be present. But, how do we check for presence of moulds, as they are not visible to the naked eye? They use a test called Dimethylene Blue (DIB) Test, where DIB solution is poured into the bottle. Voila, the moulds, if present show up as blue stains, which a trained eye can detect. Cute eh!

Would it not be interesting to invent an universal staining solution that when applied to software can make any defect visible?

Let's us continue the train of thought and see if we have any parallels.. When we insert assertions into code, it flags faulty code when violations occur. This is kinda like the staining liquid, but this is at a low level or deep inside, and requires a good programmer to lace the code intelligently with the 'staining liquid'.

As of now, we do not seem to have an approach like at a higher level that can be done with less intelligence. How about memory leak detection? Hmm.. this seems to employ the idea. Here we are looking for a specific kind of a defect (a mould), where the leak detection tool detects stains (memory leaks) without requiring any specific intelligence from developers.

Would it not be interesting to invent a universal staining solution that when applied to software can make defect visible?

Early stage design behaviour analysis

In non-software disciplines, an enormous effort is expended in design before the product is constructed out of real materials. It is indeed necessary to do so, as the cost of rework at manufacturing phase is very expensive. Think of this as "low malleability". The cost of morphing (I.e changing) at the late stage of manufacturing is very expensive. We probably believe that software is more malleable. Is it?

In other disciplines, the design is checked dynamically, by simulating behaviour whereas most often, we assess design statically via design reviews. The designs in automotive industry are subject to serious dynamic analysis of behaviour. Finite Element Analysis, Crash analysis are some examples where the dynamic behaviour is examined formally, before the product is manufactured.

Statistical sampling & analysis to build confidence

Statistical sampling is rigorously applied to check the manufactured items to build confidence in quality. The question however is - are there any activities that we do, that could be similar to manufacturing ? Software development is understood as a creative activity and therefore never compared to manufacturing.

Let us think on this.. When multiple items are being manufactured, variations can occur resulting in some items being defective. Whilst we do not seem to have a manufacturing approach, do understand that our key resource is the intellect, with the same person developing various pieces of code for a project. So is it not probable that a person could be possibly creating similar types of defects?

If so, could sampling of types of defects in various code developed by a person prove useful? Could this give an insight into our defect injection capability and therefore improve defect detection? Can this act as our personal mirror allowing us to prevent defects?

Serious focus on attributes

Non-software industries seem have evolved to a higher degree compared to ours, with strong focus on non-functional aspects- reliability, stress handling, safety, endurance, stress and usage.

We in software do focus on attributes also, however I felt that our colleagues from non software industries expend enormous effort on these and commence evaluation of these real early. On the other hand I think that we expend a considerable effort in evaluation of functional correctness. Is this because of their standards that state attributes clearly? Hmm...

Closing note...

The quest for perfection and pursuit of excellence has been the quest for mankind. Every discipline looks at the act of delivering excellence slightly differently, and examining these with open mind provides us with inspirations and innovations to a cleaner software world.

Cheers! Have a great day.

The diagnosis

May 2011

Try connecting a bunch of dots, and questions arise.
Good understanding is like mind reading.

One day Joe's roommate David was unwell. After a busy day at work, grappling with unruly software, Joe returned to his apartment and took the stairs to his pad on first floor, and set his heavy laptop back down. He opened the bag, searched for the house key and unlocked the door, welcomed by the typical darkness that he was accustomed to.

He knew the placement of the light switches very well, as he stepped to turn it on, he heard a groan. He froze, frightened by the noise; he quietly tip-toed to the light switch, turning it on rapidly. David was an owl who worked late and came in after Joe. He was shocked to see his roommate David moaning in pain.

As Joe touched David lightly, he opened his eyes lightly and said "Sorry man, it hurts, feels like I am in labour" and smiled weakly. David loved life and everybody liked his funny bone humor. Joe smiled and said "Let us go to a doctor now".

"No Joe, it was pretty bad couple of hours ago, I have taken medication and it is a lot better now. I have found a comfortable posture and would not mess with it right now. Let us go in the morning".

"Ok. Do you need something?"

"No, Thanks. " said David as he curled up and continued to take deep breaths to alleviate the pain.

Joe left, washed up and went to the bed. He was feeling bad and slightly worried. Slowly his thoughts drifted back to office. He was getting into a new project in a domain where he had never worked before. This was a new product, he was a little lost and had trouble understanding the application. There was very little written documentation and the key folks were in a different continent. Today afternoon when it was going nowhere, he had walked into his manager's office and whined.

"Well, you have to be creative and got to come up with good questions."

Joe had played with a similar application and seemed to understand some parts of the application. But he was not not able to visualize the application and its usage in its entirety. Slowly he drifted into sleep. He was awakened by sound of constant flushing in the adjacent bathroom.

"David - are you ok?" asked Joe.
"Yeah man" said David soft;y. "Guess something that I ate yesterday is really working out my system ". As David came out, he looked pale and tired, hobbled to the nearest chair, sat down and laid his head into his cupped hands.

"Boy you look terrible, guess we better go see the doc now".
"Yeah, what time is it now?"
"8:30"

In a few minutes, they were out of the apartment, hailed a taxi and was in GoodLife hospital at 9:00.

"Hi, Is the doctor in ?" Joe asked the pretty lady behind the front desk.

"No Sir, he is expected any moment. You are the first one and I will call you as soon he is in."

After a few minutes, they were ushered in to the office of Dr Holmes.

"Good morning. David, I presume you have serious pain in the abdomen with frequent vomiting in the morning" said the observant doctor surprising David throughly.

"Yes doctor, but how did you know my name?" David asked, not realizing that his name with #8 was displayed prominently on the back of his T Shirt.

David was an avid basketball fan and played for the local city team.

Dr Holmes smiled and got onto the business.

"David,when did the pain commence?"
"Couple of days ago doctor, it became intense yesterday evening"

"So what do you do David?"
"I work in an ad agency as a copywriter"

"Late nights and irregular sleep?"

"Yes doctor, the day starts late and it is pretty long"

Dr Holmes asked David to lie down and gently touched the lower part of abdomen and David winced.

As Dr Holmes continued to ask questions on family history, what he ate yesterday, prior history of sickness, Joe was transported to a different world. A reflective person that he was, he was thinking about the problem of understanding the application.

He realized that Dr Holmes was employing a pattern of questioning that was structured yet creative. Joe knew a bit of mind mapping, and realized that the doctor was seeking information on some standard aspects like family background, lifestyle, food habits, recent activities and came up with questions when he connected these aspects. He also realized that certain answers resulted in more questions.

Slowly the mist lifted and he realized that as a tester, he also needed blobs of information like customer types, types of end user, #users/type, profile of usage, key attributes, architecture, stage of development, relative ranking of the features/users, interaction between features, feature volatility, deployment environment. It dawned on him that these blobs of information and their connections would enable him to construct a "Landscape" of the system helping him to visualize the system.. It flashed on him that good questions can be asked when these connections are attempted to be established.

"Do you have any questions David?" asked Dr Holmes.

"YES!" said Joe emphatically as he thumped the Doctor's desk, his reverie broken. Realizing his faux-pas, he sheepishly looked at a confused David and an amused doctor.

"Guess you solved the problem. Good understanding requires an open mind, and connecting the dots" said Dr Holmes surprising Joe.

"Wow, you are a mind reader" exclaimed Joe. "Guess you are Sherlock Holmes!"

Dr Holmes smiled and said "Good understanding is like mind reading. Good day Gentlemen".

Joe escorted a bewildered David out of the doctor's office, said "Have a great day" loudly to the pretty receptionist, winking at the elderly lady seated past the reception.

Joe knew the answer to his problem and looked forward to a lovely day at work.

Note: Landscaping is a technique inspired by Mindmapping and is a core concept in HBT (Hypothesis Based Testing). It enables a scientific approach to questioning to aid in rapid understanding of a system.

The tale of two doctors

April 2011

External examination - Black box.
Internal examination - White box.

In a big city lived Joe, a typical urban yuppie. He was always focused on a great tomorrow. He worked very hard, partied furiously and lived a fast life.

Life was a blast, until his body decided to act up. On a Sunday morning, he woke up panting, unable to breathe, body drenched in sweat, with a dull pain in his chest. The previous evening was a blast, a celebration party thrown for his best buddy recently engaged. After an evening spent at bowling, they hit the pubs, closing each one, until they could not find one open.

A typical Sunday morning would commence at noon; today, as he was rudely jolted out of his reverie, the bright LED clock showed 7:00.He could not move his arms, it seemed to take a tremendous effort to reach out for the bottle of Evian on the table near his bed. He had read about old age diseases getting younger in these modern times, dismissed them brashly, a reflection of his supreme yuppie confidence. For once he faltered, worried seriously if he could become one of the stories. All these years, he had thought of God of as a fashion statement but today he genuinely wished to believe that God exists. For the next few seconds, which seemed like an eternity, his mind rapidly flew back over the past years on the constant abuse he had heaped on his body. For once he prayed dearly that he would do the right things, if he was excused

this time. The clock glowed 7:02, and he realized it had been the longest two minutes of his life.

At 8:55 a.m he was in the reception of GoodLife hospital for a 9AM appointment with Dr Robert Black Sr., a senior and very experienced cardiologist. He was soon ushered in, and was face to face with a severe yet friendly gentleman, a few orderly strands of golden hair on his shiny head with a piercing pair of eyes. "Mr Joe, would you please tell me your problem?" said Dr. Black. Joe described in detail his travails upon his rude awakening. An old school doctor, he believed in detailed physical examination rather than the fancy modern equipment. "Lie down on the bed and relax Mr Joe". He took his stethoscope, placed it on his chest and listened carefully. "Breathe in and out deeply now" said the doctor as he continued to hover his steth on various parts of chest. His sharp eyes showed no emotion, as he went about his job confidently. The young nurse standing next to the doctor was petite and beautiful. She dispassionately took out the sphygmomanometer, wrapped the elastic band around Joe's arm, pumped it up and watched the mercury bobbing up and down, while her other hand measured the pulse. After a minute, she looked at Dr Black and said 140/120 and 93, in a husky voice.

"Mr Joe, have you been feeling very tired at the end of day lately?"
"Yes" he said and added "It is the busy time of the year at work, a string of late nights."
"What kind of work do you Mr Joe?"
"I work in the software industry. We are in midst of building a cool application for mobile phones"
"Oh I see, you are the software guy. My nephew is in the software business too and is always racing against time."

"I guess you must be tied to the desk most of the time. Do you exercise?"
"Well doctor, the days are long and busy, I catch up on my sleep over the weekend. I try to workout in the gym over the weekends, but it is challenging"
"I see that you are smoker, do you drink? And are you a vegetarian?"
"Well doctor, I do drink and I am not vegetarian."

Dr Black was one of the most famous cardiologists in the country. He was a master at diagnosis, believed in scientific and systematic study of symptoms and their connections. He placed his gold-rimmed glasses on the table, rubbed his finger on his chin, leaned back on his cozy leather chair and his piercing eyes looked straight into Joe and said "Mr Joe, you have issue with the blood supply to the heart muscles, there seems a advanced arterial block. It is necessary that you undergo angioplasty, a procedure to relieve the constriction very quickly within the next two weeks."

Joe sat still, staring at the statement "The most amazing non-stop machine. Take care." written on a poster containing the picture of heart, which prominently on the wall behind the doctor.

Dr Black who was used to these reactions, jolted Joe out of the reverie "Mr Joe, you need to quit smoking and go vegetarian. You are young, the angioplasty procedure has a high success rate and you should be back on to active life quickly." Dr Black believed in conveying news straight, and expected patients to face reality and act on the problem. "Mr Joe, you would need to be in the hospital for 3-4 days, do decide on the date quickly. As I mentioned before, it is important that you act on this within a fortnight. It is

painless procedure and should be fairly straightforward in your case. Mr Joe, do you have any questions?"

"No doctor, I have none" replied Joe mechanically, his ability to think numbed by the turn of events. As he exited the consulting room, the receptionist, a cheerful and bubbly woman in twenties whispered "Have a good day Mr Joe" with a beautiful smile, a genuine one. It had the effect, and for a moment he felt cheerful and returned the smile, a little weakly though.

As soon as he was outside the hospital, his hand went mechanically to his shirt pocket containing the cigarette packet. "Smoking kills" said the packet loudly and he threw the packet on the wayside.

David, his roommate had just woken up as Joe returned to his apartment. "Hi, had breakfast? Got some muffins and bagels, want some?" said David. "No thanks" mumbled Joe.

After a few minutes David understood the reason for the strange behavior of his best friend. "Come on man, let us get a second opinion right away". David was one who never lost his cool and his level headed thinking in tough situations was one that helped his friends many a time.

At 10:50 they were at ValleyTech hospital for consultation with Dr James White. He had been referred by David's boss, who had undergone a heart bypass a few months at ValleyTech.

At 11:05 Joe was called in. "Good morning Joe. Please sit down. I have read your case sheet and have a few questions. Do you have any recent ECGs?

"No doctor" said Joe.

"I know your company has a yearly health check plan for all, as our hospital administers it. So have you not taken it this year?"

"No doctor, the last few months has been very hectic and I have not had my yearly checkup yet" said Joe.

Dr White was a modern doctor who relied on technology in diagnosis and treatment. A young cardiologist,he believed in seeing the 'internals' before the
scalpel touched the body. He was amazed at the advancements in radiology and made it a point to recommend a few pictures to be taken before he touched a patient.

Unlike Dr Black who believed in the power of external examination, Dr White believed in the looking at internals for diagnosis and treatment. Dr White had immense faith in scans, lab tests and preferred analyzing reports to spending time on and performing examinations on patients.

"Please get the ECG done now. I would like to see the report first. Thank you."

Joe went to the diagnostic lab in the adjoining building. When his turn came, he went inside, removed his shirt, the technician smeared jelly on his chest and proceeded to stick colorful leads at various points. In a few seconds, the needle was dancing, drawing patterns on the strip of paper. The technician looked at the squiggles on the paper with a bored expression, and after a few minutes, decided the machine has had its share of fun and

switched it off. He tore the roll of paper, scanned it intently, and then proceeded to fold it and inserted into a cover. Joe was curious to know what the squiggles meant and asked "Is it normal?".
"It seems fine,expect for a small spike here" replied the technician. He had seen hundreds of such squiggles and knew exactly as to what was normal,but he was no doctor to interpret any abnormalities.

Joe stared at the report, the squiggles held a secret that Joe was scared about. "Hey, let's go meet to the doc" now said David breaking Joe's train of negative thoughts.

"Show me the report Joe" said Dr White.
Dr White held the strip of paper and rapidly scrolled it forward and then backwards.
"Were you treated for any heart related issue when you were young?" asked Dr White.
"No" said Joe, scared to ask questions.
"Joe, there is a slight aberration in the ECG, it may be nothing to be worry about. To confirm this, I recommend that you get the 128-slice beating heart scan. This the most advanced technology for diagnosis of heart related ailments that is available in the world and we are the only hospital in this city to have this. This gives a clear picture of the beating heart and enables clear diagnosis. And also take a chest X-ray too. I will be available until 1:00 PM, get it done right away and then see me". He wrote down the lab request and handed it over to Joe.

"Hello, I need you to get '128-slice beating heart scan' done. How much does it cost? Joe asked the grim looking gentlemen on the cash counter.

He was shocked at the cost of the hi tech scan, it seemed to have enough digits to max his credit card. Joe looked at David conveying in his look "They are milking us".

Joe realized that the second opinion was going to be expensive and needed to think about this before he went on a diagnostic spree.

Joe was a professional software tester who diagnosed software for defects. He decided to chat with David over a cup of coffee to decide whether to go ahead with the expensive scan. David went to get the coffee while Joe sat down at the corner table, gazing at the birds fluttering over the little pond outside.

During this mindless gaze, staring at the chirpy birds, it suddenly flashed on him, the parallels between his profession and the doctor's. In his job, he used black box techniques that required him to examine the system externally and find defects. He relied on deep understanding of the intended behavior, observation of behavior ("symptoms") to design & refine his test cases. He had at times looked at internal information like architecture, technology, code structure to design test cases that were adept at catching issues related to structure.

His colleagues always used terms like "Black box testing" and "White box testing" and associated these to with system and unit levels respectively. Now he realized the general misconception that unit testing was white box testing and system box testing was black box testing. His train of thought was interrupted by hot coffee spilling on his shoulder followed by the shrill sound of glass breaking. "I am really sorry, hope you are ok" said the elderly woman who had tripped over the protruding leg of the gentleman

at the neighboring table and spilt the hot cup of expresso that she was carrying. "I am fine, let me help you" said Joe as he helped pick up the glass shreds.

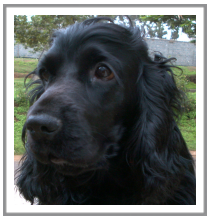
He realized that certain types of defects were better caught via "internal examination"(white box test techniques) while some are most suited to be caught via "external examination" (black box test techniques). He now understood that both of these test techniques were required at all levels to uncover effectively and efficiently.

Suddenly the diagnosis approach followed by Dr Robert Black and Dr James White became clear. As soon as David laid the steaming cup of Cafe Latte on the table, Joe had made the decision. He was not withdrawn or worried. The confidence was back and he would not let the ECG scroll spoil his fun.

De-ticking a dog – What can we learn from this?

March 2011

Purpose sharpens our intelligence.
And intelligent thinking enables curiosity to seek clarity.



I had a beautiful and adorable dog – a black cocker spaniel ("Taggy"). A wonderful medium-sized dog with lovely drooping ears, she loved to be in the midst of us. Being a house-dog, it was big challenge to ensure that she had no ticks/fleas.

Let us do a quick tour of the insect world now to understand the real bugs - Ticks and Fleas.

Ticks are wingless creatures that live exclusively on the blood of animals for three of the four stages of their life cycle. They are equipped with an apparatus called Haller's organ which senses heat, carbon dioxide and other stimuli to allow the ticks to locate the presence of an animal food source. Once found, they crawl on and embed their mouth parts into the animal's skin and proceed to suck up its blood. Ticks suck the blood and become fatter and fatter and then can fall off the dog. Ticks congregate in colonies and are generally sluggish.

Adult fleas are wingless insects, generally smaller than a sesame seed, who feed on the blood of animals. Their proportionately enlarged back pair of legs gives them an extraordinary jumping ability. Did you know that if fleas were the size of humans, they would be able to jump over the St. Paul's Cathedral in London six hundred times for three days! http://www.insecta-inspecta.com/

fleas/cat/jumping.html. Hanging on to your pet's fur with their claws, their needle-like mouth parts bite through the skin to suck up blood. Fleas are extremely agile, they move around.

Removing these bugs from a dog, particularly a hairy one is indeed a very daunting task. You never really know the number of bugs present and it's always a challenge to know if you did indeed do a good job. As a test practitioner, I was curious to learn from this activity to improve my skills to detect "software bugs".

Where did I look for the "bugs"?
I discovered that the bugs were not uniformly distributed, but did not seem random either. I understood that the bugs like to congregate between the digits of the feet, the underbelly near the legs, behind the ears, around the neck, in areas with higher density of hair and also areas that were warm and clammy. I also understood that the neighborhood of areas with dull hair, patchy skin, itchy areas were good candidates to examine for bugs.

How do we I detect (& remove) these bugs?
The typical ways are:

- Periodic shots to prevent insect multiplication and killing those that are present.
- Using a special flea dog collar to repel these insects.
- Periodic brushing of the dogs (typically at least once daily) to brush the coat to remove these.
- Parting the coat, manually looking for them and physically removing them.

During my experiments with the dog, I discovered a couple of interesting techniques to de-tick.

- One of these was parting the hair in the opposite direction allowing me to see the skin and therefore allowing a better visibility to the presence of bugs.
- When I gave the dog a bath, I discovered that the water on the dog made all the hairs stick and the bugs stood out like sore thumbs that I could manually shake out.

These were interesting to me, as techniques that made the "bug stand out".

What are the sources (causes) of these bugs?
The typical reasons for the presence of these bugs are:
- Interacting with other flea-infested dogs and picking the bugs from them.
- Dog walking in areas that could have these bugs from other stray dogs. My dog was especially susceptible to pick up these due to her long droopy ears.
- Inappropriate diet that makes the skin and the blood attractive for the bugs.

How do I know if I had done good job of "bug removal"?
Some of the observations that could indicate a good "bug removal" are a shiny coat, less itching by the dog, no bugs found on the floor. These are in addition to the ensuring al lower bug density during the later cycles of the physical bug removal.

What can we learn from this story?
That the act of uncovering defects in software/system software requires an intelligent examination of knowing what defects to look for, why these defects may be present, where to look for, how to

detect these efficiently, and what information to seek to aid the execution of the various necessary activities.

In the case of "how to uncover bugs", the dog experience shows that certain "bugs" are better detected/prevented via "periodic tick and flea injections" (i.e. internal examination - white box) while some are better detected/prevented by "flea collar" (i.e. external examination - black box).

Diligent observation of behavior (i.e symptoms 'like itching'), being sensitive to the context ('areas of dog walking', 'stray dogs', 'diet') is very important to effective testing.

That it is useful to understand where the potential defects may be present i.e. potential areas by careful observation/study.

That it is important to have rapid systematic scrubbing ('daily brushing') to lower the probabilities of defect multiplication over time.

That examination of end user results ('less itching by dog', 'shiny coat') are truer indicators of cleanliness of software in addition to the types and number of defects caught.

Interesting thoughts...
Like periodic injections given to the dog to prevent or detect the bugs, do we have something that we could inject the software with, to prevent/detect software defects? Could this be "assertions" ?

Like the water on the coat that made all the hairs stick together and therefore expose the bugs, what can we "coat the software with" so that the defects stick a better. Could "memory leak

detection", where we lace the code to surface the "memory leak issues" be an analogy?

What is the equivalent to "the act of parting the hair in the reverse direction to expose the bugs" in testing a software/system?

End note

My dog has been my mirror, she helped me reflect on software testing. I am in search for the "coat" and the "pill" that can "shake out" and "kill" the defects.

Section 3 – Realise & Evolve

This section contains articles on personal transformation. Going beyond observations and learning, to a deeper realisation of how we could do far better, resulting in the evolution to a newer self.

The stages of transformation as we grow, being in a state of bliss, expanding circles of influence, enjoying change, realising that a human uses a system, doing less to achieve more, discovering magic in the opposites and ultimately seeing beauty in testing are what these articles highlight.

Enjoy.
May you attain the ultimate!

Year++. Stay young, Have fun.

December 2012

Success requires not only how-to-do, but also how-not-to-do.
Grow old. Stay young.

As we step into the new year, it is a wonderful time to look forward to new learnings, new roles, new experiences. It is that time of the year when we make interesting resolutions. Resolutions to learn something new, to do new things, to contribute to a better world and so on.

Let us mull over this. Learning is about acquiring new competencies, knowledge to be able to do things. Knowledge does give us a sense of power, makes us feel nice. But is mere knowledge good enough ? Nah- It is only when we apply this knowledge to solve a problem successfully, do we feel a sense of true power. We look forward to application of this knowledge multiple times and only when we fail (and subsequently learn) do we feel the absolute power. This is when we become skilled.

So the FIRST transformation - KNOWLEDGE to SKILL. From knowing how-to-do to actually solving problems successfully. Note successful problem solving requires not only how-to-do, but also how-not-to-do. Don't be afraid to fail.

So as we step into the new year, it is wonderful to look forward to new learnings, successful applications and also failures. Hence there is no need for fear/tension to succeed only, this allows us to learn well.

Now onto the next stage ... Is it good enough to possess the skill to solve problems when asked to ? It is indeed nice to be called in to solve problems. It is like 'giving something' when asked for. Guess what, it feels nicer to take ownership, take responsibility rather that just be a passive bystander who can 'give' when asked for. This is absolute power, the power to change, using your skills. So take responsibility, take ownership in anything you do.

The second transformation therefore is SKILL to RESPONSIBILITY, from ability-to-do to own-the-problem. So are you thinking of what you are are going to take charge of, in the new year ?

On taking ownership to solve problems, we become good at problem decomposition, activity planning, and executing the activities using the acquired skills. The focus becomes the activities and the successful completion of each one of them. But is successful completion of activities good enough? This requires us to go the next stage. From successful completion of activities to delivering outcomes I.e delivering business value.

It is the shift from the activities to the recipient of the activities. It is about ensuring the recipient(s), (I.e end users) benefit from the solution. Delivering value to customers/end-users, rather than just successfully solving the problem. Do you what value you are delivering to your customers now? What value do you intend to deliver to your customer, your company, your team this year ?

This the third transformation from ACTIVITIES to OUTCOMES, from doing good work to delivering value to your customers, company and the team. It is when we shift from the 'doing' to the 'recipient' from the 'inanimate-activity' to the 'personal-outcome'.

Continuing the same train of questioning, Is delivering value good enough? Let us think this through.. As we continue to perform activities that deliver value, we do become busy. That is when monotony sets in, tiredness creeps in, work gets heavier and boring. And it is about getting stuff done, no more fun. Now it is high time for the next stage of transformation. The shift from doing work to having fun. The shift from 'recipient' to the 'doer'.

In addition to seeing the larger picture of activities from the recipient's view, when we immerse in the activities with the focus on the doer, time stops, you are in flow and it is pure joy. That is what work should be - Joyful.

This is the final transformation from WORK to FUN. From accomplishing for others to accomplishing for yourself. This is when you are do things for the sheer joy of doing and guess what, the outcomes not only deliver value, they are beautiful. This is what each one of us should wish in the new year, having fun, being joyful and value delivery happens. This is about staying in the present, being relaxed, enjoying every moment, being immersive and outcomes are magical. This is when you become young at heart, and experience the joy of a child.

I would love to share my story of cycling with you now. About a year ago I started cycling, with the focus on long distance rides. Initially it was about building competence on ride techniques, strength building, climbs, hydration, posture and others with focus on building endurance skills. With time and failures I became skilled in doing 100 kms rides with ease. Now I wanted to graduate to doing rides of 300 kms and beyond. That is when I discovered that it was necessary to relax and be in the moment as just focusing on time and distance outcomes were weighing me down. Just

focusing on the wheel on the front, every pedal rotation, enabled me to be in the moment, and 300 kms was fun to ride,feeling energetic even after completion of the ride. I am looking to doing 500+ long rides this year and look forward to enjoying and completing these this year.

Transformation from Stage 1 to 4.
1. Knowledge to Skill
2. Skill to Responsibility
3. Activities to Outcomes
4. Work to Fun.

Summarising ...
Competence -> Skill -> Ownership ->Activities -> Value -> Fun.

As you step into the new year, ask yourself what you want to know, what you want to be skilled in, what you want to be responsible for and what value you want to deliver. And most importantly, ensure that you have fun accomplishing these.

Have a great 2013.
Become knowledgeable, skilled, enrich others and enjoy.
Grow older - year++, Stay young & Have fun.
God bless you.

"Great expectations" – Excellence requires empathy

Oct 2012

It is not just technology, process or tools.
Remember a human uses your system.

What do we do when we find defects in software? We triage them based on severity and priority, fix the ones on the top, leaving some of the ones at the bottom possibly open.

In my recent discussions with a Japanese company, I encountered an interesting situation. The company had purchased a large software package that was customised and then deployed. During the process of customisation, they discovered quite a few bugs in the base code, a few hundred. Being Japanese, they were aghast at the quality of code and lost confidence in the software supplier.

Now I was curious. How could a global vendor ship a enterprise class financial software with fair number of open bugs? Were their tests less mature? or Did they release the system with open issues that they deemed they could fix later?

On the other hand, what kind of defects were encountered by the Japanese company? Being a stickler for perfection, were they less tolerant to even minor defects?

Digging into the details, I discovered that the number of post release defects were about 500+ and all of these were functional in nature. A majority (60%) of these were possibly related to incorrect validation of inputs, issues in the user interface and so on. Did the vendor think that these were not critical enough to fix or not so important to find (i.e. go after)?

If for a moment we do consider that the majority not possibly deemed critical, how come the customer lost confidence in the vendor? Hmm.. This is what set me thinking...

Most often the rating of a severity and priority is driven by rating scale that is typically part of the SDLC process. The rating scale is a guideline while the actual value accorded to a defect is subjective, based on the triage team. Now think about this.. The rating has to be done from the perspective of the customer, keeping in mind the different expectations of each customer. In this case, the customer being Japanese, their expectations on quality are very high and tolerance for any kind of defect very low. Hence even what is deemed as minor defect by the vendor is deemed 'not minor' by the customer resulting in low confidence.

So, it all comes to a simple thing- "Are you really sensitive?" Sensitive to what the end users expect? Being empathetic is possibly the most important social trait that is needed for being a good test engineer or a great software developer. To be in the end user's shoes, to think like them, to understand their feelings when they interact with your software is indeed very important to making key decisions in development/testing to deliver great software.

In all my interactions, I strive to look beyond the technical stuff by peering into the "person". What do they expect? How would it help them? What would they not like? The technique "Viewpoints" in HBT (Hypothesis Based Testing) is what I frequently use to viewing the system from the end user's view. It is challenging, interesting and also very frustrating as you discover how much you do not know/ understand. But, it is fun!

So in closing, if the vendor had attempted to understand the social traits of the Japanese customer - their penchant for fine details, their low tolerance for any kind of defect, their zealous "customer orientation", then the test strategy, test cases, test effort may have been quite different. And the confidence in the vendor may not have been lost.

It is not just about technology, process and tools. It is about the person who uses your system. Never forget the HUMAN side of the equation. The social side.

Empathise. Understand their issues. Build with them in mind.
See the joy when they use your system. Absolute bliss.

Have a great day.

Joy of testing – Being mindful & mindless

Dec 2013- Jan 2014

The state of being aware of only now, you realize your peak potential.
And you are so tuned, so observant. Beautiful.

Activities are important, but outcomes matter. Intellect plays a vital role in ensuring that the activities we do are indeed effective and efficient to ensure good outcomes.

Outcomes are great, but will they deliver the promise to the customer? It is about keen observations and fine adjustments that ensure outcomes deliver the value. It is about being in the 'flow'.

Let us analyze and discuss. It is the wonderful time of the year to reminisce, reflect and look forward to a joyful year. To a year filled with 'joy of testing' and delivering value to customers and the community.

The act of testing consists of a variety of activities that we perform. Some activities require intelligent thinking whilst some are repetitive. Intellect I.e Intelligent thinking plays a vital role in being effective. A logical scientific approach to understanding, hypothesizing what could go wrong, performing activities to prove/ disprove and continuous observation to constantly adjust is what makes testing challenging. And this is the basis of Hypothesis Based Testing(HBT). Being logical/rational seems interesting, but is this good enough? Is it fun? Does this not make it cold and impersonal? We may perform activities that that are indeed effective, but will they deliver value to end users? Should we not

be emotional and the keep the end users in mind to deliver value? And more importantly enjoy testing.

In additional to the intelligent activities, testing also consists of repetitive ones, that are perceived boring. And as intelligent testers, what do we do? We create a process to make the repetitive activities more efficient. In effect, be mindless. And slowly monotony sets in and we get bored. And we hate the structure/rigor imposed by the process. We perceive it to be curtailing creativity. Does being mindless make you dumb and not creative? Let us look at this differently.

Mindless requires you to become empty, so that you can be in the present, the state of mindfulness. Being in state of flow. Sheer joy. I discovered this in endurance cycling. Endurance cycling is about doing real long distances - a few hundred(s) of kilometers at one go. Like any endurance activity, this is also not about sheer physical strength, but largely about mental strength. When you ride for hours together non-stop constantly looking forward to finishing the distance (goal), it becomes monotonous, boring, saps your mental energy and exhausts you physically. And takes the fun out of cycling. What I discovered is to look just in front of the wheel, not the mile markers, and just focus on the pedal rotations. To just focus on now. In the process, I emptied my mind and became mindful, only aware of now, not reflecting on the past and blissfully ignorant of the future. Being in the present. Time stopped. I was in the 'flow'. The state of being aware of only now, and this was bliss. Sheer joy. And this is when you realize your peak potential. And you are so tuned, so observant and react to the environment in an extremely agile manner and seemingly without effort. And this is beautiful.

So when you build a process to make the repetitive things easier, or to ensure that we have clarity of what to do when, don't dismiss this as being non-creative and boring, think of this as making way for you to be uncluttered so that you can be mindful and enjoy testing, focus on end user outcomes and therefore deliver value.

Intelligent thinking is being rational, of being analytical, to process observations logically to ensure effective outcomes. Being mindful allows one to be in the present and enjoy the moment, to be very observant/ sensitive and process these observations logically, course correcting constantly and delivering value. So just be in the present and enjoy every moment.

Joy of testing - Being mindful and mindless. A feeling that is blissful. May you experience this more often in the the new year.

To the wonderful folks of this community, wish you a wonderful 2014. To a year of joy, fun and value.

Lead and ye shall grow

October 2013

Growth is not merely external, it is building inner strength.
With inner strength comes confidence and the power to influence.

Quite frequently in conversations with test professionals I come across questions that relate to career growth - "How do I grow in testing? What should I do to grow? What areas in testing should I pursue to grow quickly? ...". These are interesting conversations as there is no specific answer.

Let us dig a little deeper to understand what career growth may mean. Is growth about earning more money? About better designation? Or more 'power' in terms of larger project/team size...? I bet your answer will be "All of these". Hmm, these are outcomes of growth, but the question still remains -"what is growth?". Is growth deeper knowledge? Or skills/abilities? Or is it the confidence to get anything done?

I would like to believe that the last one is most appropriate. It is about having the confidence of getting anything done. If you can handle stuff that is more fuzzy, difficult, constrained, large, with huge expectations, then you grow faster and higher. It is not just doing work, it is about getting work done. It is about being a leader.

Reflect. A young baby needs support to get things done, as he grows, he manages to be self-sufficient. As he becomes a young adult, he is able to manage others and later as an adult when he establishes his family, he leads, not just manage. As you grow this is what happens : Managing yourself --> Managing others(or things)

--> Leading (others/things). The ability to "do", "manage doers" to "leading doers" is what defines growth. Initially it is about managing, later evolving into leadership.

Leadership is about influencing others. It is about believing in yourself and not really care about what others think. A far cry from being dependent on what others think about you to figure out if you are good. It is about having belief and confidence in yourself. It is not about 'toeing' the line, it is about leading. It is not just agreeing to other's view, it is about healthy arguments to put forth yours views/thoughts.

Bodily growth cannot happen when you just eat more, it happens only when the food is digested and assimilated. Likewise, it is not about just knowing more, it is about deep understanding aided by reflection that allows us to assimilate. This results in more than just knowing, it is about forming heuristics as when to/not-to apply in way similar to purging unwanted stuff. This results in one become more confident in the application of the knowledge.

Growth is evolution and evolution is "Add/Modify/Delete". Visible external growth is "Add", the accumulation of mere knowledge. Real 'internal' growth is "Modify/Delete", the assimilation that changes the internals, and purgation, the "Delete" that discards old views/ ideas, strengthening you from inside. Growth is not merely external, it is building inner strength.

With inner strength comes confidence and the power to influence. To influence the project team we work with. To influence the product we are building. To influence the company where we work. To influence the customer who uses our product. To influence the test & software engineering community.

The expanding spheres of influence :
Individual "I" --> Project team "US" --> Product "OURS" --> Company "US & OURS" --> Customer "THEM" --> Community "WE".

Now you do not care about what others think. You have grown. Stuff happens. You do not 'just do actions and get things done'. You deliver 'business value'. You feel good. And possibly get noticed. Anyways you enjoy what you have done. And the wonderful outcomes of growth happen.

As another year comes to a close shortly, it is natural to think how we have grown and chart the path for the future. It is not just merely knowing new technology, techniques, domains, tools, it is about how to use the power of knowledge to influence all around you. This year, think not on the outcomes of growth, chart out the plan to influence.

"Lead and ye shall grow".

Change. Continuous motion. Cadence.

September 2013

Constant adaption is wonderful as it results into continuous fluid motion.
A wonderful feeling of aliveness.

We all know that change is constant. We also know that adapting to change is hard. We resist change.

An adaptive system responds. Responds rapidly to change. In nature, this is key to survival. And also key to delivering high performance.

Change is challenging and worrisome. We resist because we are typically afraid. Afraid of possible bad outcomes. Afraid of the risk it puts us in.

To adapt to change requires information. Information that we can use confidently to respond well.

Let us look at how we respond when changes are done to software. When a change is done to software, we always get into a mode of regression. Worrying about 'will what was working-before continue to work as the way it was'? And then also worrying about 'how might the changes affect other stuff negatively'? And our response is to use our experience to guide us to choose a subset of the larger set of test cases to regress in case they are non-automated. Or choosing a large set of test cases that we think is necessary to be run always and automating them.

Let us go a little deeper. A change is done i.e. code or configuration has been modified to change the behaviour. Let us understand as to a behaviour is. A set of conditions that the entity should satisfy. So when a change is done, new conditions may have been added, existing conditions modified or deleted. So our worry is if those parts of behaviour that still are the same i.e those combination of conditions that are unchanged function the same way as there were before. So the information that we should elicit therefore is about the what conditions are new/modified/deleted. Note that conditions may be of various types - data related (syntax/format, size, rate, type etc), behaviour related (functionality), load-handling related, performance-related, security-related and so on.

The second aspect of the change is the domino effect. What other parts of the system can the change impact? And what kind of impact - functional behaviour, performance, security etc. To know what other parts of the system may be affected, we need to have a good understanding of the linkages between the various requirements/features/system-elements i.e what requirement/feature is connected to what requirement/feature. Having understood the linkages, the next step is to analyse/qualify the linkage based on the change i.e. what conditions of the linked element would be affected by the change done?

Let us look at this a little differently, from the perspective of validation. What we are keen to understand is the fault propagation ability of a change i.e. how could a fault in the system created due to a change propagate via the linkages and ultimately cause a failure? So correlating the changes done to an entity to the potential types of defects (of that entity) that could be irritated allows us to choose the appropriate test cases to regress that

entity. Proceeding further, analysing as to how an irritated potential type of defect could propagate along the linkages i.e affect the other elements (by irritating its potential types of defects) allows us to select the appropriate test cases for the linked entity. Interaction matrix and Fault Propagation are some of the techniques used in Hypothesis Based Testing (HBT) to stay on top of changes to ensure optimal and effective validation.

Using information about conditions, potential defects types, linkage and fault propagation allows us to respond to changes well and adapt testing constantly.

Constant adaption is wonderful as it results in continuous fluid motion. A wonderful feeling of aliveness. And that is what nature is about. Continuous morphing, resulting in improvements. As software evolves, enjoy it. Adjust and adapt- the test strategy, the plan, test cases and tooling.

In cycling/running there is an interesting concept of cadence. In the case of cycling, cadence is about how many times/minute you rotate the pedal. When you move from a plain terrain to a climb (change), the speed will drop. So the natural response is expend more power by pedalling harder to maintain the speed. The challenge is that heart rate rises rapidly and it is a just a matter of time when this becomes difficult. This is where cadence comes handy. Instead of expending higher energy, shifting to lower gears and rotating the legs (pedals) faster results in maintaining the same speed on the incline. So it is typically recommended to spin at a higher cadence to compensate for the change. Correlating to what we do, my take is that cadence is about continuous motion to enable a good response to the change. High cadence implies nimbler movement and using information about the terrains as we

ride enables one to respond well. Now I like climbs and look forward to climbs!

In running the concept of cadence is about the number of steps we do in a given time. Rather that run with long steps, it is suggested that you take small steps, and take more steps during a minute. Taking small steps and more of these per minute allows one to use energy optimally yet achieve higher speeds.

Summarising the rate of work ("cadence") enables us to respond to "change" well (of course we need good information) enabling a "continuous motion" (that is ultimately a state of bliss).

This being a special issue on "Women in testing", I cannot resist connecting these concepts to this theme. In my opinion women handle changes and adapt very well. They manage multiple things(spouse, kids, family, house, work,...) and each of these 'change' (most often cause problems!) continuously and they seem to juggle these very well. To all of you there, my salute!

So next time when a feature changes, enjoy it. Develop a high cadence process/suite that allows you to respond rapidly and enjoy the continuous motion.

Not more, but no more

August 2013

Focus. Think. Analyse deeply.
It is about brain, not brawn.

When we want to do significantly better, the tendency is think that we need to do more. More tests, more cycles, more automation etc. It is only natural to think that we need to accomplish more.
This puts a lot of pressure, and the challenge is to ensure that we respond to this well, not buckle. In the next few paragraphs, I am going to argue that it is not about 'doing more', but it is really about intense focus to 'do no more'. That is, remove fluff to 'do no more'.
Seems contrarian eh ...

Let me illustrate this with a story. When I got into long distance endurance cycling where the minimum distance of a brevet is 200 KMS, I thought it is about building stamina, fitness and got to doing more - More rides, more time on the saddle, more speed, more distance. The challenge I faced was that it was becoming difficult to ride for a longer time as I was constantly looking at the clock and the milestone marker, and sadly both seemed to move slowly, it was frustrating. That is when I decided to do something different - to focus only on the front tyre and ride. What this did to me was amazing. It allowed me to see only short distance in the front and not worry about all the 'mores' I.e distances, time, speed. Suddenly I was in the present and could achieve higher saddle time by 'doing no more'. The trick was to sharpen the focus, in this case this was the distance to look at and doing just what you need do, no more.

When you want to find 'more' defects, focus on the objective of what types of defects you want to uncover. Focus on examining information that can give you a better clue to the types of defects. This goal focus enables to come with just enough (no more) and not necessarily more test cases.

When we want to deliver higher quality software, it is not about doing more testing. It is about focussing on the types of defects that could creep during the entire SDLC, and then figuring how to detect earlier or prevent and therefore 'do no more' rather than 'do more'.

When an application has matured and is in the maintenance phase, number of defects uncovered by the (typically) large set of test cases is low. Here again it is not executing more, it is focusing on which test cases have become 'immune', and then focussing on those that have higher 'potency' based in the potential defect types that could be irritated by changes done. Here again this is another case of 'do no more'.

When we want to shorten release times, it is not only about more automation to execute more to shorter time. It is also about focusing on those types of defects that are more probable and executing those I.e 'do no more than necessary'.

'Doing no more' requires us to focus, to think and analyze deeply. It is about using the brain, not the brawn of (more) hard work. And this is the fun part.

The pressure situations, where the objective is to accomplish more is wonderful. Because this is when we can focus on 'what to

remove' and 'do no more' rather that 'physically work hard' to 'do more'. Personally I love these situations, as they help me focus sharply and drop all the fluff.

So when somebody tells you to do more, correct them so that they understand this as 'no more'. Pressure situations demand brevity (do no more) and this is possible only when you focus, be in the present, think well so that you can do only those that matter.

The doctors in the emergency room face intense pressure. They need to make split second decisions and respond quickly. Brevity is key here. Saving a life is not about doing more, it is about just doing what is required -'No more'.

Well, I better shut up.
No more.
May you accomplish more.

Yin and Yang in testing: The magic is in the middle

June 2013

Zero is infinity.
When you are empty, unattached, you are filled with bliss.

As we mature we see more opposites. Is the objective of testing to find more issues or prevent them by being more sensitive? Is behavioural information more suitable that structural information to design test cases? Is automated testing superior to finding issues?

There are very many things in the discipline of testing that seem contrary, like the Yin and Yang, creating a constant tension as to 'what-to-choose'.

This explores the idea the various opposites and outlines the view that "The magic is in the middle". That it is not a tussle, but a perfect state with the opposites balancing each other.

Let us look at the act of understanding ... Should we know the entire system completely, in detail ? Will this bias us? Should we attempt to understand the whole at one go or should be understand only as much required and defer the rest to later ? What should we understand - how the system should-work/is-working or how it is architected/built ? And what is the granularity to which we need we understand - precise or approximate? And do we read the spec and understand or play/experiment and figure it out ?

As we can discern from here, it is 'neither this nor that', but something in between. What the middle is, is based on experience.

And experience is in knowing the governing variables that enable decision making. Some are these are:

1. State of the product/application (brand new vs. existing)
2. Area of complexity (internal I.e structural OR external I.e behavioural)
3. Degree of availability of information.

And application of general principles like: onion peeling, experiment in addition to trust, approximate and iterate.

Let us to explore the opposites in test design:

‣ Should we have more test cases or have focussed & specialised test cases?

‣ Which information is more suitable to design test cases - external behavioural information or internal structural information?

‣ Should we take the viewpoint of end user or deep technical viewpoint to design?

‣ And what should the distribution of positive & negative test cases look like?

And once again, it is neither the extremes, it is the middle! And what are the governing variables here ? Some of these are: quality level for which we are designing (early stage building blocks, or end user use cases), #inputs to combine to generate test cases, type of defect we are looking forward to uncover.

Moving onto the aspect of test execution- Is automated testing better that using intelligent humans? Should we do more cycles of execution or less ? And finally should we test more often I.e scrub more.

The middle is based on some of these governing variables - is the objective stability/cadence (health check), defect yield, feature addition frequency.

Lastly examining the process of validation, how heavy/light should the process be, how disciplined versus creative should we be, how much of thinking (a-priori) versus observation & learning should we adopt, and how much compliance & control vs freedom? A process is most often mistakenly understood as an inhibitor of creativity, when the real intent is to 'industrialise' the common things, ensure clarity in interactions so that we can channel our energies from the mundane stuff to higher order to deliver performance. Doing this requires us to be mindful and be in the present and 'enjoy the doing'. Some of the variables at govern this are : complexity, error proneness, degree of availability of information, fun factor.

Now the big question - is the objective of testing to detect issues or to enable us to have a heightened sensitivity so that we can prevent issues ? You figure this out.

Summarising, there exist opposites like the yin and yang. The trick is find the middle that is harmonious, requiring the right mix of two the polar ends. Not via a compromise, but a wise choice of the middle discovered by using by a set of governing variables gained with experience. The harmony that experience in the middle when we balance the two ends is the the "magic".

On a philosophical note, zero is said to be infinity ! This means when you are empty, unattached, you are filled with bliss! And they are not really opposites as we perceive, the magic is the middle- 'the bliss'. As a long distance endurance cyclist, I discovered the magic, the bliss when I focused on the front wheel rather than the mile marker during long rides lasting multiple hours. Not to be worried about the distance to destination, or judge the performance in the distance covered, but to enjoy the cycling. The

magic is to be in the present, to be mindful. Not the past, nor the future.

The magic is in the middle.
Recognise it, exploit it harmoniously to deliver high performance. Cheers.

Aesthetics in software testing

Jan-Feb 2011

Craftsmanship is the ultimate expression of good work.
Beauty touches our heart, utility satisfies the mind.

Software testing is typically seen as yet another job to be done in the software development life cycle. It is typically seen as a clichéd activity consisting of planning, design/update of test cases, scripting and execution. Is there an element of beauty in software testing? Can we see outputs of this activity as works of art?

Any activity that we do can be seen from the viewpoints of science, engineering and art. An engineering activity typically produces utilitarian artifacts, whereas an activity done with passion and creativity produces works of art, and this goes beyond the utility value. it takes a craftsman to produce objects-de-art, while it takes a good engineer to produce objects with high utility value.

An object of beauty satisfies the five senses (sight, hearing, touch, smell and taste) and touches the heart whereas an object of utility satisfies the rational mind. So what are the elements of software testing that touch our heart?

Beauty in test cases

The typical view of test cases is one of utility– the ability to uncover defects, Is there beauty in test cases? Yes I believe so. The element of beauty in test cases in its architecture - "the form and structure".

If the test cases were organized by Quality levels, sub-ordered by items (features/modules) then segregated by types of test, ranked by importance/priority, sub-divided into conformance(+) and robustness(-), then classified by early (smoke)/late-stage evaluation, then tagged by evaluation frequency, linked by optimal execution order and finally classified by execution mode (manual/ automated), we get a beautiful form and structure that not only does the job well (utility) but appeals to the sense of sight via a beautiful visualization of test cases. This is the architecture of test cases suggested by Hypothesis Based Testing (HBT).

Beauty in understanding

One of the major prerequisites and for effective testing is the understanding of the product and the end user's expectations. Viewed from a typical utility perspective, this typically translates into understanding of various features and intended attributes. To me the aesthetics of understanding is the ability to visualize the software in terms of the internal structure, its environment and the way end users use the software. It is about ultimately distilling the complexity into a simple singularity–to get the WOW moment where suddenly everything becomes very clear. It is about building a clear and simple map of the various types of users, the corresponding use cases and technical features, usage profile, the underlying architecture and behavior flows, the myriad internal connections and the nuances of the deployment environment. It is about building a beautiful mental mind map of the element to be tested.

Beauty in the act of evaluation

Typically testing is seen as stimulating the software externally and making inferences of correctness from the observations. Are there possibly beautiful ways to assess correctness? Is it possible to instrument probes that will self assess the correctness? Can we

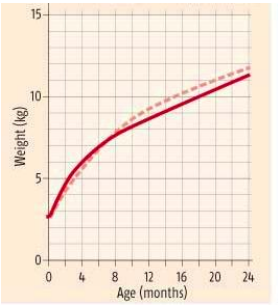
create observation points that allow us to take better into the system? Viewing the act of evaluation from the aesthetic viewpoint, can possibly result in more creative ways to assess the correctness of behavior.

Beauty in team composition

Is there aesthetics in the team structure/composition? Viewing the team collection of interesting people – specialists, architects, problem solvers, sloggers, firefighters, sticklers to discipline and geeks etc. allows us to see the beauty in the power of the team. It is not just about a team to get the job done, it is about the "RUSH" that we get about the structure that makes us feel 'gung-ho', 'can-do anything'.

Beauty in reporting/metrics

As professionals, we collect various metrics to aid in rational decision-making. This can indeed be a fairly mundane activity. What is aesthetics in this? If we can get extreme clarity on the aspects that want to observe and this allows us to make good decisions quickly, then I think this is beautiful. This involves two aspects–what we collect and how we present these. Creative visualization metaphors can make the presentation of the aspects of quality beautiful. Look at the two pictures below, both of them represent the growth of a baby.



The one on the left shows the growth of a baby using the dreary engineering graph, whereas the one on the right shows the growing baby over time. Can we similarly show to growth of our baby (the software) using creative visualization metaphors?

Beauty in test artifacts

We generate various test artifacts - test plan, test cases, reports etc. What would make reading of these a pleasure? Aesthetics here relates to the layout/organization, formatting, grammar, spelling, clarity, terseness. These aesthetic aspects are probably expected by the consumers of these artifacts today.

Beauty in the process

The test process is the most clinical and the boring aspect. Beauty is the last thing that comes to mind with respect to process. The aesthetic aspects as I see here is about being disciplined and creative, being detailed yet nimble. To me it is about devising a process that flexes, evolves in complete harmony with external natural environment. It is in the hard to describe these in words, it can only be seen in the mind's eye!

Beauty in automation and test data

Finally on the aspect of test tooling, it is about the beautiful code that we produce to test other code. The beauty here is in the

simplicity of the code, ease of understanding, modifiability, architecture and cute workarounds to overcome tools/technology limitations.

Last but not the least, aesthetics in test data is about having meaningful and real-life data sets rather than gibberish.

Beauty they say, lies in the eyes of the beholder. It takes a penchant for craftsmanship driven by passion, to not just do a job, but to produce object-de-art that appeals to the senses. As in any other discipline, this is very personal. As a community, let us go beyond the utilitarian aspects of our job and produce beautiful things.

Have a great day.

My Musings

When you want to understand, 'storify'. Describe.
When you want to solve a problem, state rules. Prescribe.

Syntax is a great guide. A guide who provides you the rules.
Rules that enable you to stay on the path of clarity.

Good understanding is not about knowing more.
It is about discarding what is not necessary.

With no rules, comes immense freedom and power.
And with great power comes responsibility.

Logic dissects a problem from outside and then solve it.
Experience helps you understand the problem from inside and then solve it.

Good behaviour is about compliance to rules.
*Good test cases find loophole**(s)** in these rules.*

Write less. Communicate more.
Think deeply. May the light flow into you.

It is not just seeing what is present.
It is about seeing what is absent.

Quality is about meeting expectations, properties of a system.
And testing is about assessing how well these properties have been met.

Trust and proof are on diametrically opposite poles.
Trust you can do well, Prove that you have done well.

Run. Run fast. Run long. Run well.
Eat. Eat enough. Eat right.

When you are potent, the world is at your feet.
With time, everything becomes impotent.

Tame complexity by breaking it down.
If you cannot chew, prepare to be eaten.

Everything is complex.
The trick is to make it look simple.

Goodness is holistic.
Beautiful outside. Strong inside.

Structure is what holds the various parts of the system together.
And a robust structure is key to good functional/non-functional behaviour.

Do we measure the distance before we back up a car? No!
We approximate the distance and refine continuosly. And it is natural.

Good questions matter more than the availability of answers.
It is not about what you know, it is about you do not know.

A good mirror reflects what you are.
And what you see changes you.

It is not about knowing the outside.
It is about changing the inside.

A fault when irritated results in a failure.
A domino of faults when perturbed results in a catastrophe.

Innovations happen when you look outside your discipline.
Being constantly curious is what makes life enjoyable.

Try connecting a bunch of dots, and questions arise.
Good understanding is like mind reading.

External examination - Black box.
Internal examination - White box.

Purpose sharpens our intelligence.
And intelligent thinking enables curiosity to seek clarity.

Success requires not only how-to-do, but also how-not-to-do.
Grow old. Stay young.

It is not just technology, process or tools.
Remember a human uses your system.

The state of being aware of only now, you realize your peak potential.
And you are so tuned, so observant. Beautiful.

Growth is not merely external, it is building inner strength.
With inner strength comes confidence and the power to influence.

Constant adaption is wonderful as it results into continuous fluid motion.
A wonderful feeling of aliveness.

Focus. Think .Analyse deeply.
It is about brain, not brawn.

Zero is infinity.
When you are empty, unattached, you are filled with bliss.

Craftsmanship is the ultimate expression of good work.
Beauty touches our heart, utility satisfies the mind.

~~~~~~
~~~~~~

Ashok is the Founder & CEO of STAG Software, a pure play test boutique. Do what you love and Love what you do" is what he lives by. Passionate about solving problems, he has been focused on building on ways to scientifically test software, He is the architect of Hypothesis Based Testing (HBT), a personal scientific test methodology.

"Simple is complex' - he revels in taming complexity and enjoys the learning and discovery it offers. A strong believer in opposites, the Yin and Yang, he strives to marry the western system of scientific thinking with the eastern system of belief and mindfulness.

He is an alumnus of Illinois Institute of Technology, Chicago and College of Engineering, Guindy. Keen on endurance sports, he is an avid long distance cyclist and a half marathoner.

He can be reached at ash at stagsoftware dot com, @ash_thiru on Twitter

They say we are creating history with this.

How about finding it all by yourself?

Tea-time with Testers

A FREE monthly magazine for Software Testers and everyone else who cares about quality.

Get your free copy from www.teatimewithtesters.com