PROFESSIONAL TESTER

Essential for software testers

SUBSCRIBE
It's FREE
for testers

October 2014 | £ 4 / € 5 | v2.0 | number 29

Including articles by:

Gregory Solovey
Alcatel-Lucent

Llyr Jones
Grid-Tools

Staffan Iverstam
QualityMinds

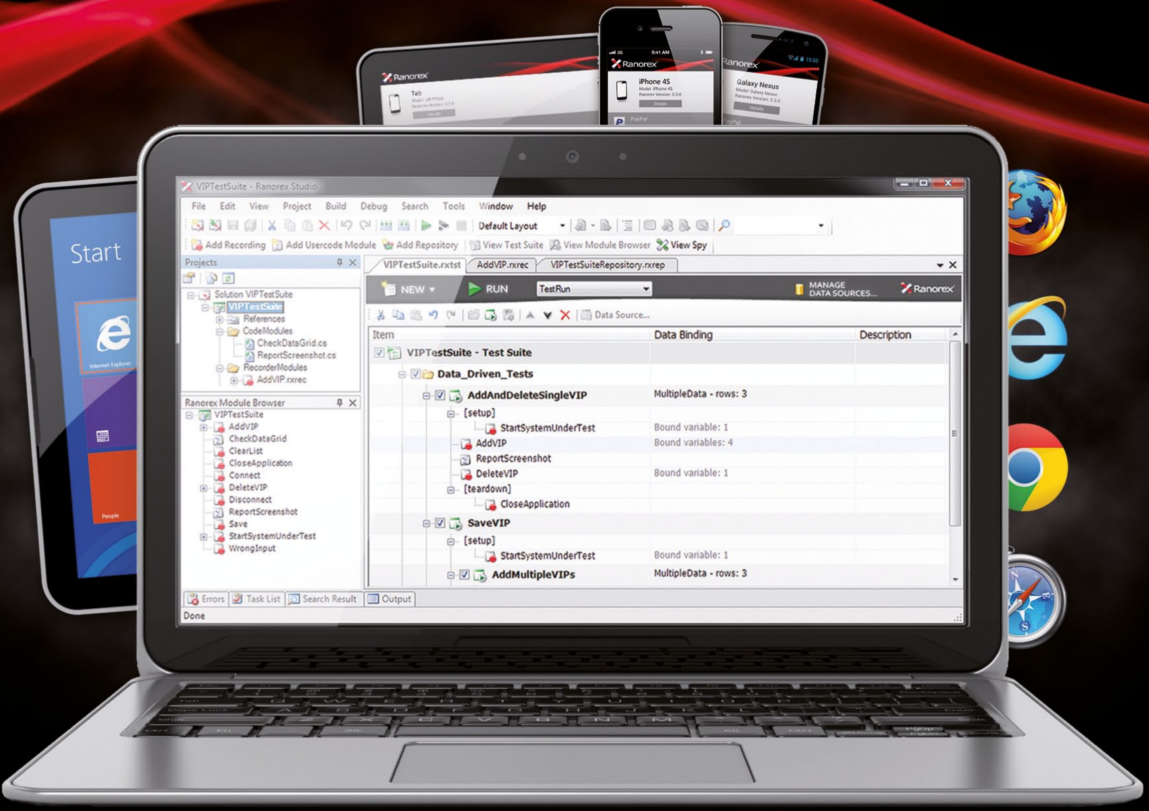
Reshama Joshi
L&T Infotech

Stefan Patry
NORIZZK.COM

Sakis Ladopoulos
INTRASOFT International

Ranorex®

Automated Testing of Desktop. Web. Mobile.



Robust Automation

Broad Acceptance

Seamless Integration

Quick ROI

Why Use Ranorex

www.ranorex.com/why

Award-winning test automation tools provide seamless testing of a wide range of desktop, web and mobile applications.
Android | iOS | HTML5 | IE | FF | Chrome | Safari | WPF | Flash/Flex | Silverlight | Qt | SAP | .NET | MFC | Delphi | 3rd Party Controls | Java

The revolution will be in high definition

Imagine a tester is given a test basis and applies a test technique to it to design and specify tests.

Imagine another tester, who has no contact with the first tester, is given the same test basis and applies the same technique and specifies exactly the same tests.

You have just imagined the future of test techniques, therefore of testing, therefore of software development.

It will be achieved by better, that is more rigorous, definition of test techniques. Taking the most fundamental of them, equivalence partitioning, as an example: telling us to choose an arbitrary input within a partition is not sufficient. Tell us exactly how to choose for any partition. More difficultly, tell us exactly how to partition input and output of any test object.

Similar genericization can be achieved for all the test techniques currently known and for those yet to be known but proposed or imagined as by contributors to this issue. Its many positive effects will include enabling development of perfect automation of test design and truly comparable measurement of test effectiveness.

This hard but essential and long-overdue work will eventually make avoidable defects a thing of the past, to be remembered and regretted but never repeated.

The initial goal is that every defect detectable by known techniques is detected at the first opportunity. With that achieved actual innovation will, at last, be able to start.

Edward Bishop
Editor

Contact

Editor
Edward Bishop
editor@professionaltester.com

Managing Director
Niels Valkering
ops@professionaltester.com

Art Director
Christiaan van Heest
art@professionaltester.com

Sales
Rikkert van Erp
advertise@professionaltester.com

Publisher
Jerome H. Mol
publisher@professionaltester.com

Subscriptions
subscribe@professionaltester.com

Contributors to this issue
Gregory Solovey
Llyr Jones
Staffan Iverstam
Reshama Joshi
Stefan Patry
Sakis Ladopolous

Professional Tester is published by Professional Tester Inc

IN THIS ISSUE

SUBSCRIBE
It's FREE
for testers

Visit professionaltester.com for the latest news and commentary

From test techniques to test methods

by Gregory Solovey

Good enough test design in very little time

PT proudly presents **Gregory Solovey**'s hierarchical, step-by-step test design methodology

Some say test design should be considered an art or craft. However here I will try to present it as a systematic process by showing that all functional tests needed for any software product release can be built in a matter of days, based on architecture and requirements documents.

Applying test techniques to the complexity to which they are defined, or too exhaustively, in a quest to be able to detect exotic and unlikely defects is impractical. These defects when they do occur are usually the result of poor coding practice and not testable at all. Basic application of the well-known techniques to define simple test design methods is sufficient to detect almost all implementation (that is, coding) defects which could occur.

Even products containing millions of lines of code need not be daunting, because the test effort is not proportional to the size of the code, but to the size of its independent components. The layered architecture of modern software, with lower layers providing services for higher ones through APIs, follows that paradigm.

For example, an embedded system consists of drivers, OS, middleware and applications. Each of these comprises independent services and features. Each software release is described by a set of incremental requirements and architecture documents. Before development starts, these requirements are refined to sufficient detail. Now test planning and test generation start. Presenting each requirement as a software model (an expression, an algorithm, a state machine, etc) and using a known method to build test cases should be a straightforward, routine process.

Putting theory into practice

Theoretical test design techniques take into account all possible levels of object complexity, for instance the presence of multiple defects that can mask single ones, lack of output interfaces, etc. In addition, they are intended to provide a quick and exhaustive test design algorithm for objects with thousands of elements. In most real-world circumstances, testers do not need to resort to such "muscle" methods.

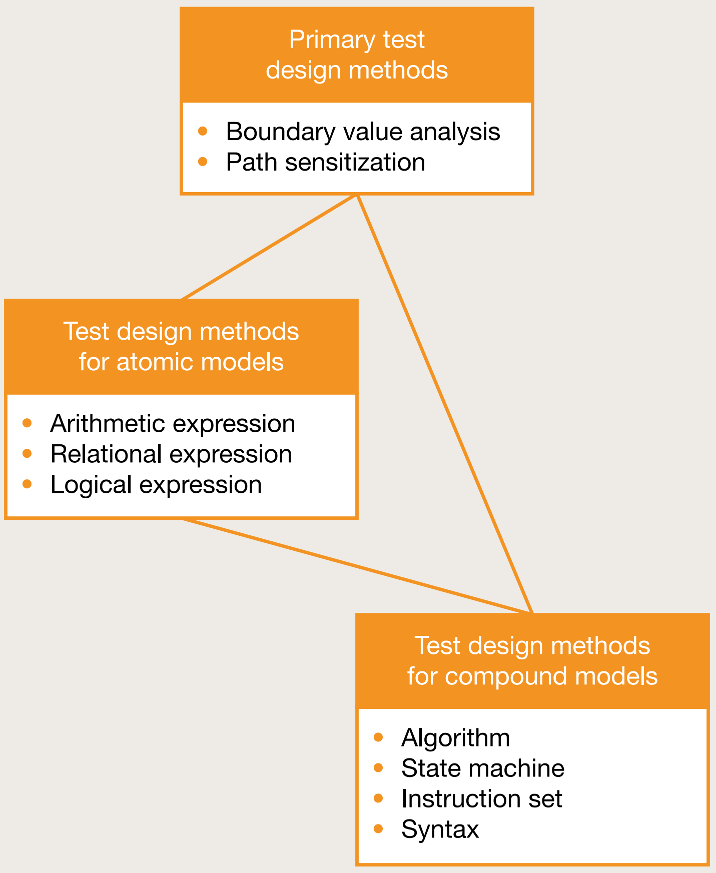


Figure 1: derive schema of test design methods

First, the requirements are represented by much smaller scale models, containing less than one hundred elements. Second, a developer responsible for the requirements' implementation can easily improve insufficient object controllability and observability by, for example, adding a “show” function or a CLI command that works directly with the API.

Object models
A requirement is usually written in business terms, which is not the best format for writing test cases, because it does not include formal definitions of possible errors. Formal models (such as condition, algorithm, state machine, etc) on the other hand use formal definitions of error classes and test design methods that guarantee defect coverage. The tester should ensure that each requirement is presented as a formal model prior to beginning test design.

Defect models
Test cases have to identify all the implementation errors for the formal models.

This article will deal with the defect model sometimes called ‘symbol swap’ where, for example, a variable name is inadvertently replaced with a constant or another variable of the same type. This cannot be detected by a syntax analyzer.

Defects other than symbol swap, for example the presence of incorrect arithmetic, relational or logical operators, are expected to show up as symbol swap elsewhere in the code.

Primary and derived test design methods
I consider the primary methods to be boundary value analysis (selecting test cases at the edges of equivalence partitions) and path sensitization (identifying logical paths from the defect to the output that will ensure the effect of the defect if present will be propagated to the output). These two methods are the basis for defining test design methods for arithmetic, relational and conditional expressions, that is atomic models, and can be combined to define further test design methods for compound models such as algorithms, state machines, instruction sets and syntaxes (see figure 1).

Arithmetic expressions
An arithmetic expression may consist of numeric constants and variables and arithmetic operators and can be evaluated. If A and B are arithmetic expressions, then so are A + B, A ÷ B etc.

Because arithmetic expressions can evaluate to many results (for non-binary values) a small number of test cases can detect all possible defects in them. Two cases are needed to verify an arithmetic expression that does not include a division

a	b	c	output
10	30	50	7.75
-30	21	14	-23.5
5	6	6	Error message

Figure 2: test cases for a + 45 ÷ (b - c)

a	output
3	F
4	F
5	T

Figure 3: test cases for a > 4

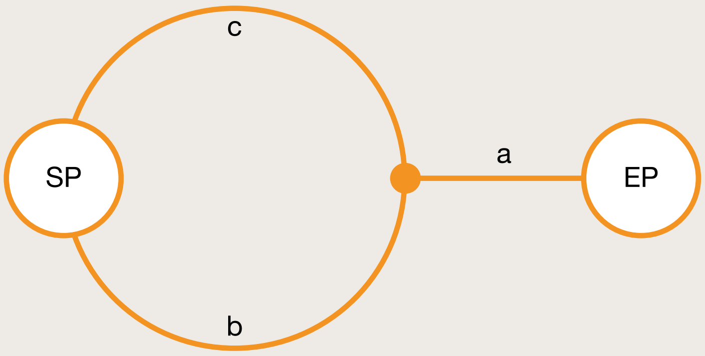


Figure 4: Danilov graph of F=((b or c) AND d)

operator. Each variable in an arithmetic expression needs to be presented by two different values in two test cases to make sure that it was not accidentally replaced by a constant or another variable. For each division operator an additional test case is needed to test the outcome when division by zero would occur.

For example, the arithmetic expression a + 45 ÷ (b - c) requires three test cases as shown in figure 2.

Relational expressions
These consist of arithmetic expressions separated by relational operators. If A and B are arithmetic expressions then A > B, A == B, A <=B etc are relational expressions.

The branch coverage approach, where each relational operator can be true or false, obviously leads to two test cases per operator: for example, for the expression a > 4, test cases a = 3 and a = 5 would be derived. However this misses possible defects such as a >= 4 instead of a > 4. For this reason BVA should be used to generate three test cases, including one on the boundary, as shown for this example in figure 3.

Logical expressions
These consist of logical constants and variables. Relational expressions are a subset of logical expressions. If A is a logical expression then so is NOT A. If A and B are logical expressions then so are A AND B, A OR B etc.

Experience has shown that using Victor Danilov's Graph model (see *Identifying test for a nondirected graph* (sic) by V. V. Danilov and B. I. Filimonov, Avtomat. i Telemekh., 1973, no 7, pp157–161) simplifies the test case creation greatly.

First, remove all global logical negations using de Morgan's Laws:

- NOT (A AND B) is the same as (NOT A) OR NOT B
- NOT (A OR B) is the same as (NOT A) AND (NOT B)

Now represent each variable as an edge (line) and, starting from the most deeply embedded brackets, represent each OR by parallel connection of edges and each AND by sequential connection of edges. For example, F = ((b OR c) and d) is represented by the graph in figure 4.

Now consider the value of each of the Boolean variables. For the case where a Boolean variable is FALSE, remove that edge from the graph. If after removing all FALSE edges a path from the start to the end point still exists, then F = TRUE.

Let's call any set of Boolean variables that, when their value is TRUE, form a pathway from SP to EP, a "path". For example the path cd means c = TRUE and d = TRUE. Another path is bd.

A 'cut' is the opposite of a path, that is a set of variables which, if all of them are FALSE, makes F = FALSE, is there is no path. For example, cb is a cut because if c = FALSE and b = FALSE then F = FALSE. Another cut is d.

Identify the minimum set of paths and cuts that cover all graph edges. Now write one test case for every path, by assigning TRUE to every edge (variable) that is on it and FALSE to all the others. Inversely, write one test case for every cut by assigning FALSE to all the variables it includes and TRUE to all it does not.

As a more realistic example, consider the following functional requirement:

One LED below each T1/E1 port indicates one of the following: (a dial feature card (DFC) is up AND (an alarm is received on the associated T1/E1 port, indicating 'loss of alignment (LOA)' OR 'loss of

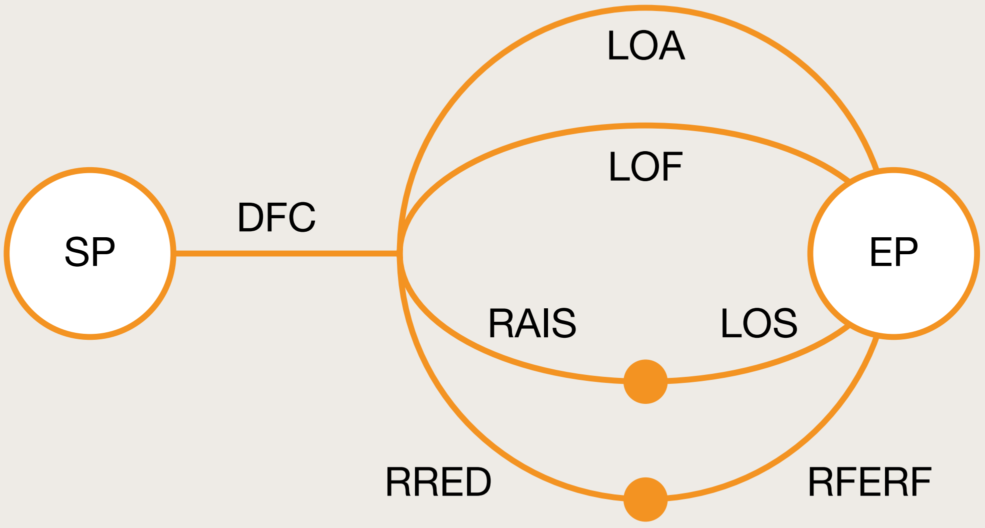


Figure 5: Danilov graph of DFC AND (LOA OR LOF OR (RAIS AND LOS) OR (RRED AND RFERF))

	DFC	LOA	LOF	RAIS	LOS	RRED	RFERF	output
path 1	T	T	F	F	F	F	F	T
path 2	T	F	T	F	F	F	F	T
path 3	T	F	F	T	T	F	F	T
path 4	T	F	F	F	F	T	T	T
cut 1	F	T	T	T	T	T	T	F
cut 2	T	F	F	F	T	F	T	F
cut 3	T	F	F	T	F	T	F	F

Figure 6: test cases for DFC AND (LOA OR LOF OR (RAIS AND LOS) OR (RRED AND RFERF))

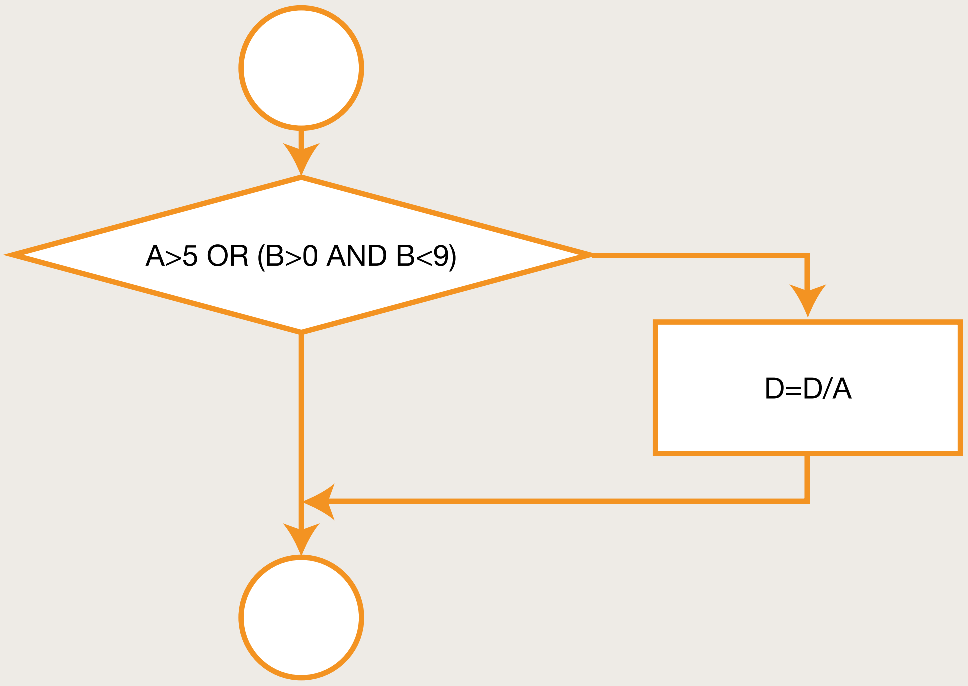


Figure 7: flowchart of IF (A>5 OR (B > 0 AND B < 9)) THEN (D = D ÷ A)

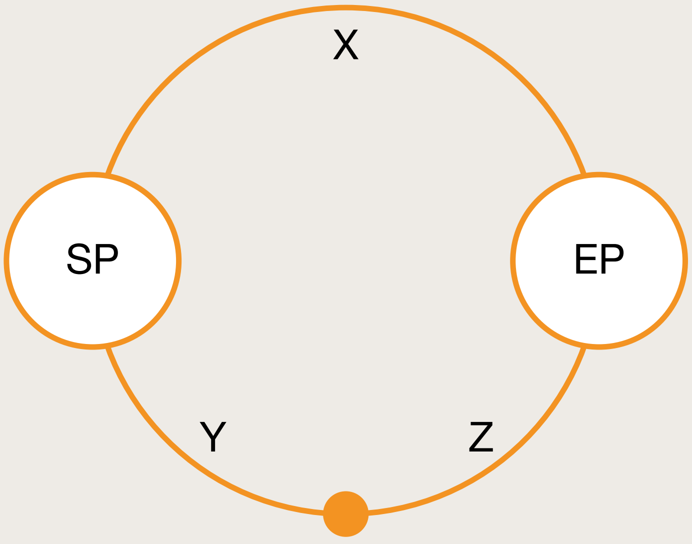


Figure 8: Danilov graph of X OR (Y AND Z)

	X	Y	Z	output
path 1	T	F	F	T
path 2	F	T	T	T
cut 1	F	F	T	F
cut 2	F	T	F	F

Figure 9: test cases for X OR (Y AND Z)

		X	Y	Z	A	B	output
1	path 1	F	F	F	4	0	F
2	path 1	F	F	F	5	0	F
3	path 1	T	F	F	6	0	T
4	path 2	F	F	T	4	-1	F
5	path 2	F	F	T	4	0	F
6	path 2	F	T	T	4	1	T
7	path 2	F	T	T	4	8	T
8	path 2	F	T	F	4	9	F
9	path 2	F	T	F	4	10	F
10	path 2	F	T	F	0	8	ERROR
	cut 1						
	cut 2						

Figure 10: test cases for A>5 OR (B > 0 AND B < 9))

	A	B	D-input	D-output
1	4	0	6	6
2	5	0	6	6
3	6	0	6	1
4	4	-1	6	6
5	4	0	6	6
6	4	1	8	2
7	4	8	6	1.5
8	4	9	6	6
9	4	10	6	6
10	0	8	6	ERROR

Figure 11: test cases for IF (A>5 OR (B > 0 AND B < 9)) THEN (D = D ÷ A)

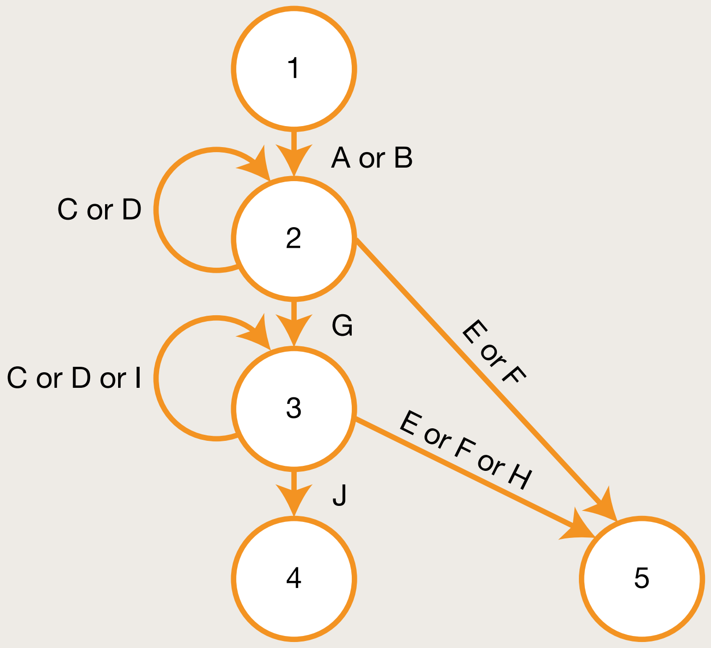


Figure 12: state transition diagram

multi-frame (LOF)' at the local or remote node OR received 'Alarm Indication Signal (RAIS)' AND 'Loss Of Signal (LOS)' OR 'Receive RED Alarm (RRED)' AND 'Far-End Received Failure (RFERF)'))

This can be written as the logical expression DFC AND (LOA OR LOF OR (RAIS AND LOS) OR (RRED AND RFERF)) and graphed as shown in figure 5. Compare the seven test cases generated (figure 6) with the 128 which would be generated by simple permutation of all the Boolean variables.

Algorithms
An algorithm is a set of functional and conditional blocks and connections among them. Each functional block contains arithmetic expressions and each conditional block contains logical expressions.

As we have already seen, most structural test techniques (including statement, branch, path etc coverage) do not guarantee detection of all possible implementation defects. That can be achieved by combining the two test design methods described above: first, BVA to create test cases for each expression in the algorithm, then path sensitization to ensure that the effect of failure of any of those cases is propagated to the output.

Consider the simple algorithm IF (A>5 OR (B > 0 AND B < 9)) THEN (D = D ÷ A), shown as a flowchart in figure 7.

The three relational expressions, A > 5, B > 0 and B < 9, are encapsulated into a logical expression. Let's name them X, Y and Z. Each relational expression requires three test cases:

	A	B	C	D	E	F	G	H	I	J	transitions
1	T	F									1
2	F	T									1
3	T	T									1 2
4	T	T	F	T							1 2 2
5	T	T	T	F							1 2 2
6	T	T			F	T					1 2 5
7	T	T			T	F					1 2 5
8	T	T	F	F	F	F	F				1 2 2
9	T	T					T				1 2 3
10	T	T					T		F		1 2 3 3
11	T	T					T		F		1 2 3 3
12	T	T	F	F			T		T		1 2 3 3
13	T	T			F	T	T	F			1 2 3 5
14	T	T			T	F	T	F			1 2 3 5
15	T	T			F	F	T	T			1 2 3 5
16	T	T	F	F			T	F		F	1 2 3 3
17	T	T					T			T	1 2 3 4

Figure 13: test cases for STD in figure 12

	command	output
1	createMailbox name, max message size, max messages	boxID, ERR
2	findMailbox name	boxID / NO, ERR
3	getMsgBuffer message size	bufID, ERR
4	writeBuffer bufID, size of data, data to write	OK, ERR
5	sendMsg boxID bufID	OK, ERR
6	getBuff boxID	message
7	releaseBuff bufID	OK, ERR

Figure 14: instruction set for a messaging service

	command	output
1	createMailbox	createMailbox
2	findMailbox	createMailbox findMailbox
3	getMsgBuffer	getMsgBuffer
4	writeBuffer	getMsgBuffer writeBuffer
5	sendMsg	createMailbox getMsgBuffer writeBuffer sendMsg
6	getBuff	createMailbox getMsgBuffer writeBuffer sendMsg getBuffer
7	releaseBuff	createMailbox getMsgBuffer writeBuffer sendMsg getBuffer releaseBuff

Figure 15: minimum set of sequences for instruction set in figure 14

rule.test#	test
1.1	Verify that an error message is output if element ucSet does not contain element uc
1.2	Verify that one element uc can be processed
1.3	Verify that two elements uc can be processed
2.1	Verify that an error message is output if element ucSet does not contain attribute name
2.2	Verify that an error message is output if element ucSet does not contain attribute startTime
2.3	Verify that attribute 'startTime' is in format 'YYYY-MM-DD HH:MM:SS'
2.4	Verify that attribute name cannot be presented more than once in an element uc

Figure 16: tests to detect defects in implementation of syntax rules for an XML file

- X (that is, $A > 5$) requires test cases A = 4; A = 5; A = 6
- Y ($B > 0$) requires test cases B=-1; B=0; B=1
- Z ($B < 9$) requires test cases B=8; B=9; B=10

Substituting these new symbols for the relational expressions, the logical expression they form can be written as X OR (Y AND Z) and represented by the graph in figure 8, analysis of which gives four test cases (figure 9).

Now consider the arithmetic expression in the algorithm, $D \div A$. it requires three test cases such as: D = 6; A = A1. D = 7; A = A2 and D = 7; A = 0 (where $A1 \neq A2$).

All these test cases, for the relational expressions and the arithmetic expression, are now combined and propagated to the output (figure 10). For example, path 1 makes the output depend on the value of X and therefore enables us to verify the relational expression $A > 5$; path 2 makes the output depend on the values of Y and Z, enabling verification of $B > 0$ and $B < 9$. Notice that, in this example, there is no need for specific test cases to deal with the cuts: cut 1 is achieved by test case 4 (also 5) and cut 2 by test case 8 (also 9).

Furthermore, test cases 3 and 6 are equivalent to two of the three test cases for $D \div A$. We need only add test case 10, for the division by zero case. The complete, fully-specified set of test cases is shown in figure 11.

So far we have not considered loops in code execution. However this is easily achieved by a standard structural technique: for the logical expression which controls the number of iterations, based on the boundary conditions, design tests that cause the code within the potential loop to be executed zero times, once, the maximum number of times, and that maximum plus one.

State machines
A state machine is represented by a set of states it can be in and a set of input conditions that trigger those transitions.

The occurrence of a transition may also generate outputs: in fact for testing this is required, since otherwise it may be very difficult to identify with certainty the current state. Developers should be able to implement a function to output the identity of every state as the transition to it occurs.

Transitions are triggered by events, which are simple Boolean inputs. So, transitions are logical expressions and test design for them is simple using the method already defined. Paths comprised of a series of transitions are subject to the same rules as a Danilov graph and can be subjected to path sensitization in a similar way.

Figure 12 shows a state transition diagram for an embedded system. The capital letters denote inputs, that is events, that should cause the transitions. Figure 13 shows the 17 test cases required, in conjunction with the test cases for each of the transitions itself, to detect all possible single defects in the implementation of the state machine. This is not the minimum set to do that, because it has been expanded and arranged in order that each test starts from the initial state, 1. This easy, common practice helps test execution by allowing it to continue even when a defect is found, by continuing to execute tests not affected by the defect, including those that need to start from states that are not reached due to the defect, and helps incident resolution by keeping the result of each test independent from that of others.

Instruction sets
An instruction set is a collection of commands each specified by its name, its input parameters, and a specification of how its output results should be derived from its input parameters. Figure 14 shows an example, for a messaging service.

For a command to be testable, it must be possible to initiate its execution, and observe its outputs, using an external interface.

For each command create a unique 'macro-instruction'. A macro-instruction consists of commands that allow to initiate a test stimulus from the external interface and to verify the result of its execution.

The macro-instructions can be ordered based on the nested relationship to assist with incident resolution (figure 15).

The tests themselves are then designed using the methods already described, selected according to the content of the model of the command under test.

Syntaxes
A syntax is a set of rules that describes the various elements of objects and what are the correct structures in which they can be arranged. Many of the rules are defined based upon on the definition of preceding rules.

Test design should start from the base (independent) rules and continue following the order of rules. BVA is applied to each element, then the other methods to each of the various expressions encountered.

Consider the following fragment of the syntax definition for an XML file:

1 The 'ucSet' element contains one or more non-empty closed elements 'uc'

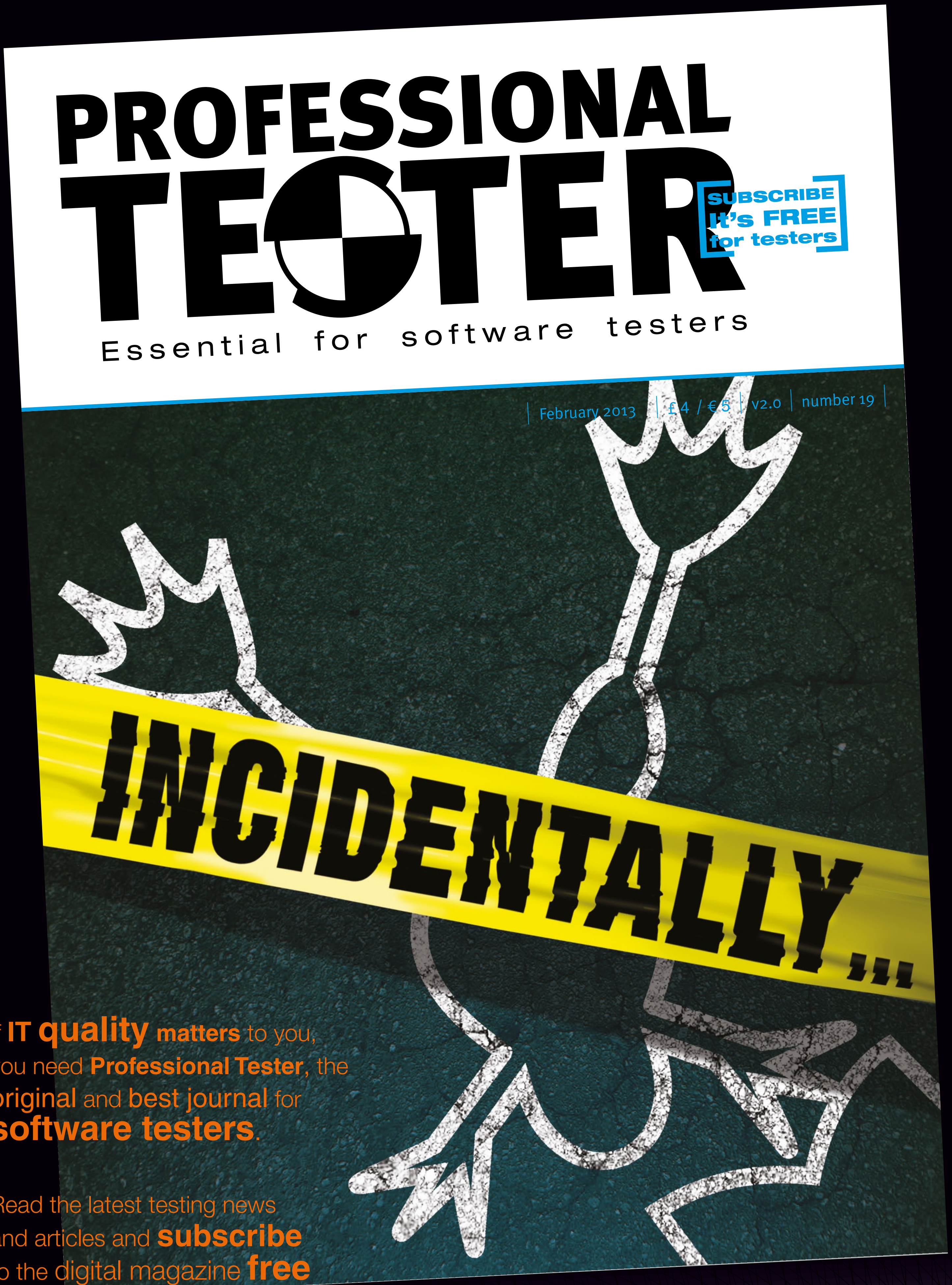
2 The 'uc' element contains the following mandatory attributes: 'name', 'startTime', and a non-empty closed element 'tcSet'

3 The 'tcSet' element contains one or more non-empty closed elements 'tc'

4 The 'tc' element contains the following mandatory attributes: 'name', 'status', 'executionTime'

Figure 16 shows the test cases required to detect implementation defects for rules 1 and 2. Only defects that could not be detected by an automated XML syntax checker should be covered by test cases ■

Gregory Solovey PhD is a distinguished member of technical staff at Alcatel-Lucent. He is currently leading the development effort of a test framework for continuous integration


PROFESSIONAL TESTER
Essential for software testers
SUBSCRIBE It's FREE for testers
February 2013 | £4 / €5 | v2.0 | number 19
INCIDENTALLY...

If IT **quality** matters to you, you need **Professional Tester**, the original and best journal for **software testers**.

Read the latest testing news and articles and **subscribe** to the digital magazine **free** at **professionaltester.com**

professionaltester.com

Keep it not real

by Llyr Jones

Enforcing testability as a test technique

Llyr Jones explains his vision for the post-agile world

Has agile fallen into the same trap as did sequential development methodologies, by itself becoming too rigid and prescriptive, preventing rather than helping teams to work the way that suits them and their current project?

Sometimes strict adherence to whatever variation of agile has been adopted does not seem the best approach but teams are forced to use it anyway. The "post-agile" idea suggests that the pendulum has swung too far and its oscillation needs to be dampened: that a new 'middle way' is needed.

So what will be the important techniques in post-agile testing? I have argued, in PT and elsewhere, for test design techniques based on flowcharting: that is, using mathematical algorithms to work out the optimal set of test cases and guarantee coverage. Applying flowcharting also shows that not all programs are testable. More recently, I have encountered people who cling to the idea that non-testable programs are sometimes inevitable, even acceptable. Also, I was shocked to read in the conclusion of Gregory Chaitin's book *Meta Math!: The Quest for Omega* (Vintage, 2006, ISBN 9781400077977) his view that *'Experimentation is the only way to "prove" that software is correct. Traditional mathematical proofs are only possible in toy worlds, not in the real world. The real world is too complicated'.*

In this article I will argue that only testable programs should be written.

Maths and meta-maths

Euclid's fifth postulate, known as the parallel postulate, states that any two lines either intersect or are parallel and is an example of what modern mathematics calls an *axiom:* something accepted as true without the need to prove it. More than 2,000 years after Euclid, mathematicians proved that the parallel postulate could not be proven from Euclid's other four postulates so they tried removing it and considering them without it. That led to the discovery of hyperbolic geometry, the cornerstone of Einstein's relativity theories.

Later mathematicians started to try to develop rigid frameworks of axioms so that all of mathematics could be put on firm logical ground. In the same way, these axiomatic systems were then analyzed to see what happened when axioms were added and removed. This field is the meta-mathematics referred to in the title of Chaitin's book. In 1931, the first provably unprovable statement, Hilbert's continuum hypothesis, was discovered, rocking mathematical philosophy to its core.

More work was done to figure out the extent of the problem. This did not go well: Kurt Gödel's two incompleteness theorems, published in 1931, stated that no matter what system of arithmetic is used, it is always possible to find a problem that is unprovable.

The parallel (no pun intended) between logical systems and computing systems is obvious. Here we have a logical framework for solving mathematical problems, and they have been shown to be incomplete. Similarly, we have a logical framework (the operators in the instruction set of a CPU) for computing. The question arises: can we prove that any program is testable?

Unfortunately, the answer, in the abstract, is no. Working at around the same time as Gödel, Alan Turing posed his *halting problem* which asks whether it is possible to prove that, assuming infinite infrastructure and resources to support its execution, a program will eventually terminate. In general, it is not. Even worse, it turned out that this could not be proven for most programs. Chaitin finally joined the two problems together, showing that Gödel and Turing's work are equivalent. Furthermore, Turing's work highlights important equivalence between the provability of mathematical problems and the testability of programs.

Defining testability
From this brief account of the spectacular failure of logical systems in mathematics, one could be excused for believing that there's no point to them. If logical systems cannot prove the majority of things, then why bother?

The same question was asked by the early proponents of agile when they exposed the pitfalls of the V model but then, unfortunately, simply replaced one set of problems with another. Mathematics never made this mistake because it simply got on with the problems that could be solved. That is why I do not share Chaitin's doom-and-gloom prophecy for testing. One must bear in mind that there are still infinitely many problems that can be solved by logical systems! There are more problems which cannot, but that is no reason not to use logical systems where they work.

Any program can be written with the basic constructs of a programming language: statements, arithmetic operators, conditions (that is, if-then-else or case constructs) and loops. Similarly, mathematics is founded on a set of axioms or primitives that define what is meant by a set and what operations can be done on a set, for example *Zermelo–Fraenkel set theory with the axiom of choice.* ZFC cannot prove its own consistency, but that does not stop it being used by most as the foundation of the whole of mathematics.

Similarly in testing, the trick is in realizing that, if one restricts scope to programs that can be provably testable, the problem goes away! This is where flowcharting comes in. The structure of a flowchart depicts the four basic constructs. If a program cannot be flowcharted, it must use something other than those and as such may not be testable.

As Gregory Solovey also points out in this issue of PT, our notion of testability must take observability into account. To be detectable, all the possible logic defects must be manifested at output: the process of making them so can be called path sensitization.

We also need to deal with the halting problem.

So we have our definition:

A program is testable if and only if its Turing machine representation can be flowcharted, is deterministic and is always-halting, and all possible faults can be sensitized to output.

I am working on proving this and if successful will share the proof with PT readers. In the meantime however I am convinced, intuitively, it is correct. As I have shown in previous PT articles, all possible use cases for a graph can be computed, and techniques such as functional equivalence used to reduce the number of test cases. Note also that every deterministic, always-halting Turing machine has a state-diagram representation which is functionally equivalent to a flowchart.

Enforcing these criteria during software design means we can develop any software possible, yet stay within Chaitin's toy world, avoiding the real world and its uncertainty. Everything we develop will be *entirely* testable!

It works even for multi-threaded programs. The difference is that multiple states exist concurrently (that is, threads change state independently of one another), causing difficulty at the join points where threads interact: this is where race conditions and locking issues are manifested. But if each thread is always-halting, then the compound program is also always-halting. So if the program is testable then it always halts and will never enter an infinite loop due to threads locking each other out!

Agile doesn't work, but being agile might
I began this article by bemoaning the fact that agile has become rigid, then went on to advocate a one-size-fits-all solution to the problem of software testability. This may seem contradictory, but can be explained by considering technological and human factors separately. The technological factors are less complex than the human ones and are, for now, restricted by the technology platforms on which they

are employed. The nature of computer programs – their composition from, only, the four fundamental constructs – will not change in the foreseeable future.

Human factors on the other hand always change, and it is a fundamental truth that the more complex a problem is, the greater the number of solutions it requires. Therefore the only chance of achieving agility is to be capable of employing many different tools including, where needed, the ones agile has recently been trying to throw away.

I suggest therefore that development philosophy – in the sense of choosing development methodologies – should be focused on human factors. The technological factors are dealt with by the notion of testability I have described. I believe it can work because it has worked for mathematics. Many people hope software development will become mature by learning from older disciplines such as engineering and manufacturing. Mathematics – at least 2,500 years old – has something fundamental to teach us too ■

Frequent PT contributor Llyr Jones is a senior developer at Grid-Tools (http://grid-tools.com)

2015 software quality days

EXPERIENCE THE VALUE OF QUALITY

20th – 22nd January 2015
Hotel Savoyen, Vienna

The leading Conference on Software Quality, Requirements, Testing and Agility

More than 50 lectures and workshops
Testing and automation, agility, tools, requirements, process improvement, embedded systems and more
Key subject 2015: **Software and Systems Quality in Distributed and Mobile Environments**

Program and tickets online
↗ www.software-quality-days.com/shop

BUY ONLINE NOW

Keynotes:

Rex Black

Skynet Has Arrived
Will Ubiquitous Connectivity Give Us Convenience or the „Terminator"?

Dr. Gernot Starke

Software ändern – aber richtig
Worauf es bei Evolution, Wartung und Änderung von Software wirklich ankommt.

Dr. Armin Wolf

Waren die Medien früher besser?
Warum ist Qualität im Journalismus wichtig und worunter leidet sie?

Change for change’s sake

by Staffan Iverstam

Generating and implementing Ideas for innovations

Staffan Iverstam explains how to stop novelty wearing off

Test improvement has always been a favourite PT topic. To many people, the term is simply short for ‘test process improvement’ which can be summed up roughly in four steps:

1 Analyse the present situation

2 Decide on a few areas to improve

3 Change the way you work in those areas

4 Go to 1.

In TPI this usually involves trying to move closer to the way of working your chosen model or standard recommends, which may require anything from a small tweak (better) to major reorganization or even policy change. Either may be difficult because the models and standards available tend to be too open to interpretation, are not necessarily suitable for your situation and are not necessarily that good anyway.

In order to get new ideas you need to think outside the box. TPI models and standards can easily keep your improvements inside the box.

So how can the test organization invent *new* ideas aimed at its specific needs and aims – that is, new test techniques – that are not only bad copies of standards or models? To avoid confusion between terms, I prefer to call initiatives to make that happen, and the adoption and integration of the new ideas that emerge *test innovation.*

My own work, as a tester closely involved with development, requires a constant stream of new ideas and solutions, especially for test case and test data creation. That has led me to become interested in other organizations that seem successful at innovation, test and otherwise: for example Samsung, Gore-Tex, Toyota and, especially, Google and its management model as described by Annika Steiber in her book *Googlemodellen* (Verket för innovationssystem,ISBN 9789187537127, in Swedish). I am not saying these are necessarily good examples of how to develop, operate or test, but of how to innovate.

In this article I present some practical ideas I hope readers will want to try to change their processes in order to drive and encourage innovation, leading in turn to larger, continuing beneficial change rather than just process polishing. I believe this is how the test techniques of the future will be invented.

The innovative organization
Let us start at high level. How did Google become so innovative? It was not an accident. Its management model was developed from its beginning with the focus on creating an innovative company: not only hiring creators and innovators, but creating a culture for creative and innovative work, including expectation of continuous improvement, generosity and openness between colleagues. The role of managers is to guide and act as role models, allowing their staff to make decisions themselves. Individuals are asked to spend only 70% of their time working for the business: 20% may be spent on personal projects related in some, possibly vague, way to the business and 10% is for personal projects that have, as far as anyone knows at the time, nothing to do with Google at all.

The innovative test organization
A lot of emphasis is placed on efficiency. Consider the number of users, the amount of dependency in the code-base and the rate of change achieved. Examples of how include:

- early adoption of continuous delivery
- making clear that quality is “owned” by developers and the focus of testing is to help developers achieve quality
- maximizing test automation, meaning that most testers are also good programmers
- terms such as unit, integration and system, which suggest (arguably) sequential phases are not used. Instead, testing is referred to as either small, medium or large. The implication for the very concept of software lifecycles is clear.

Sometimes Google invents: often it adopts early: sometimes it is the only adopter. What is important is it has moved quickly to using, not just talking about, methods and tools that are fit for its needs. And it has never stopped, but continues to pursue new ones and so improves faster and faster.

The innovative test manager
The contexts of modern testing work are very varied. Every tester knows how radically different it is to test a new-build versus an existing product. Real-time integrations, external services, multiple platforms and very many other factors all change the game frequently too. It’s vital to realise that the possibilities for innovation – where and how – are just as varied. To be useful an idea does not need to be generic or to relate only to the test process. Even if it seems to be only for a specific, obscure or unusual context, it may when analysed and developed prove to be important in other, or even all, situations. This is especially true when the aim is to improve how the test process interfaces with another activity, by changing either or both.

In the same way, it must be easy for everyone involved to innovate for testing: not just testers but the developers, clients and specialists with whom they work. All ideas must be captured for development and potential implementation. It must be remembered also to continue to manage ideas which it has been decided not to implement at this time. They will be the foundation for more ideas.

The innovative tester
One of the many marvellous things about the human brain is, I think, its ability to autopilot. We do many things, having learned to do them, without thinking about them much or at all. In most of life that is a good thing, but its effect on our work as testers is not good. Being used to one way of logging into an application tends to make us forget that there are other ways. Long experience of writing test plans can make your plans too narrow; being accustomed to dealing with developers can reduce your effectiveness at challenging them; and so on.

Even worse, the autopilot brain is brilliant at looking for similarities and finding patterns. This is good in that it allows us to use earlier experience to solve new problems, but bad in that it tends to stop us finding new ways to solve problems new and old. Faced with a new test challenge, we may intend to find the perfect solution, but the chances are we will just use the one we usually do.

Being aware of these self behaviours is the first step to becoming more innovative. The second is to make a deliberate effort to break them, to force your brain to think new.

Self-consciousness is not always pleasant: trying the following creativity techniques might make you feel silly. But please try them anyway: no idea is silly if you use it and it works.

Random entry
This involves imagining that the thing – entity or activity – you are thinking about is something else entirely, then considering what that would make diffierent.

For example, suppose you want to help people better to read your test plan. Pretend it is not a test plan but something else, a random entry. Let’s say an aeroplane! The information in the plane would be like passengers and the potential readers would be like destinations. But some destinations would be more important and popular than others, and some might not even have a runway big enough for your plane. So maybe you need more than one plane of different sizes? Or maybe you can drop some passengers by parachute, or have them use rail or road links from some large destinations to others? Now go back to the test plan. What might these ideas mean for how to create and communicate it?

Challenge and alternatives
Take a fact you believe to be true, or a thing you believe to be good, and consider what you would do if it were a lie, or absolutely terrible. For example, if you are having trouble implementing a test because of the test data it needs, think "suppose this test is terrible, will never run, has no potential to detect defects and is not worth this trouble. And yet I still have to deliver the assurance this test intends to. How can I do that?".

The answer can be reached by using a mind map, writing down the alternatives in branches. Include good as well as bad alternatives: do not let self-criticism stop you from reaching potentially good outcomes. Now let each alternative expand into more branches. You will find many possible ways of implementing the test.

It's important not to fall into the trap of using this technique to attack things of which you are actually suspicious. That will only make you less able to deal with them. Pick things you actually believe in and then subvert that belief. In the example above, the ideal would be if you yourself were the author of the test, and very proud of it.

SCAMPER
Some of the activities making up this popular creative thinking technique – substitute, combine, adapt, modify, purpose/put to other use, eliminate, reverse and rearrange – appear to have their origin in formalized brainstorming. Their intention will be obvious to readers and plenty of information is available elsewhere so rather than go into detail here I will just recommend Amanda Graham's YouTube video at http://youtube.com/watch?v=G8w0rJhztJ4 (retrieved 21st October 2014 1230hrs UTC).

I also recommend the mind map technique, as described earlier, used in two main ways: firstly as a way of expressing a problem visually and so getting a new perspective on it, and secondly as an effective way to keep notes of your own brainstorming. The brain has a habit of making associations. These can take you to ideas, but it is easy to miss the ideas in the unstructured stream of thoughts. Mind maps help you to benefit from the associations by retaining the ideas.

The innovative test improvement process
I started this article by discussing, very simply, the cyclical nature of test process improvement. Here is a more detailed suggested strategy for including test innovation in that activity (see figure 1).

1 Strategy: decide how much time will be devoted to attempting to innovate, how often it will be done and how it will be evaluated.

2. Focus: select an activity or area you want to improve: for example, test reporting, repeating manual testing, coverage of requirements etc etc. This can be quite difficult especially if you feel many areas have a lot of potential for improvement and affect one another. In that case there is nothing wrong with starting with random choices: write some possibilities on pieces of paper and draw one from a hat. (My company QualityMinds has developed preprinted cards specifically for this purpose; please get in touch if you would like one a pack)

3 Form a group and have a meeting: you need people with knowledge about both the work as it is done now and different ways it could be done. This is one reason why it is so worthwhile to attend testing events and presentations, read magazines articles etc even if they are not about your domain. At the meeting, explain the purpose, maybe present one of the creative thinking techniques above, and ask participants to work *individually and alone* to produce ideas

4 Have a second meeting: a day or two later, ask each participant briefly to present one or two of his or her ideas, then everyone to elaborate and build upon the ideas of others. Near the end, ask for rough consensus on which ideas are most interesting, taking into account the feasibility of implementing them

5 Assign a next step to each idea: for example "implement now", "develop", "investigate", "archive" etc. Also note the identity of the idea's originator and the date it was first suggested

6 Continue to develop ideas to move more of them towards "implement now" and them implement them. For each implementation, assign a person to act as its driver: motivating for it, managing it, capturing its results and informing about them. Progress should be measured for each idea implemented; not quantitatively, but qualitiatively based on the perceptions and opinions of those people affected by it.

7 Evaluate results and inform about them

8 Go to 1.

The innovative management model
A "management model" in this context means the way top management leads and motivates staff and allocates resources. Obviously this varies a great deal between organizations. But research indicates that truly innovative ones often have the following elements.

- Desire is obvious. it is communicated frequently and very clearly that innovative work is actively sought, that time and money are being allocated to it, and that more of both is available

- Work is decentralized. Small groups that can make their own decisions are more innovative. Too many routine obligations make them less so

- Micromanagement is avoided: managers stay out of detail but instead trust the group with the decentralized task

- A creative environment is maintained. The most important factor is that the staff wants to innovate. HR ensures that staff know they are considered the most important asset

- Creative and entrepreneurial personalities are present. All personalities are valuable to innovation but you can't start a fire without a spark (© B. Springsteen)

- Long term initiatives aimed at nourishing innovation and creativity are started frequently, then supported and carried through

- There is cooperation between units. The entire organization works together with clear shared ideas and goals. Ideas for better communication and collaboration are especially valued and implemented often, even on experimental basis

- Innovative work and new ideas are lifted up. It is made clear that the good thing is that the idea is produced even if it is not implemented, a decision that usually is not made by its originator and must not be allowed to prevent subsequent originations.

Are you testing's next revolutionary? It is the nature of all good testers to complain about the status quo, to be objective and to love realism to the point of pessimism. The great testing innovations have not come from fighting against these attributes, but from using them. I hope these ideas for innovation will help that trend to continue, in the spirit that change is dangerous but lack of change is fatal ■

Staffan Iverstam (staffan.iverstam@qualityminds.se) is a test manager at QualityMinds (see http://qualityminds.se)

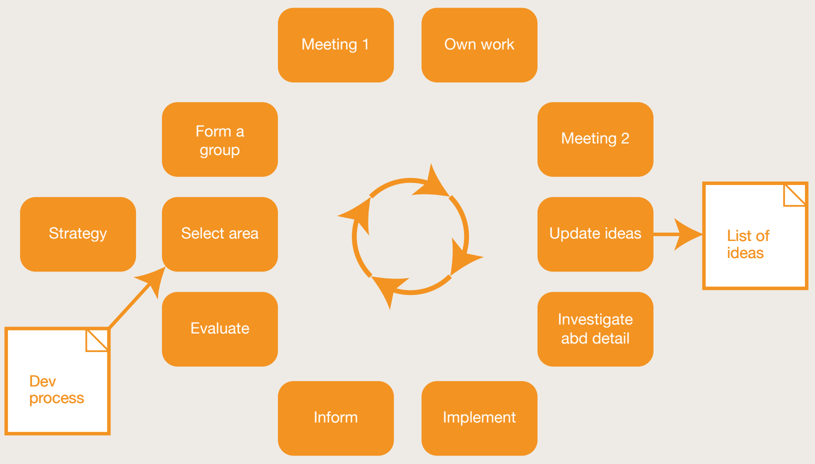


Figure 1: example innovative test improvement process

http://www.xqual.com

Manual & Automated Test Management

Only **$179** per user

What really burns

Professional Tester may consider me a 'book burner' (see http://professionaltester.com/news/Article.asp?id=318) because I have some concerns about standards. As of August 29th, I have not yet signed the _Stop 29119_petition (http://ipetitions.com/petition/stop29119 retrieved 28th August 2014 1700 UTC) because I have not yet been able to read the standard it opposes. I've tried to, but it seems I have to pay CHF 178 for just the first part Concepts and definitions (http://iso.org/iso/catalogue_detail.htm?csnumber=45142 retrieved 28th August 2014 1700 UTC). I'm already wondering if this is primarily a way to make money rather than to help people test better. Worse yet, it isn't even detecting my locality, making it harder for me to pay, even if I wanted to.

However my main concern – and that of many, I believe – is about enforced implementation. Could I be required to use and comply with a standard I have never seen before in order for my employer (a financial institution with many B2B relationships) to win a contract? Suppose, for example, the standard requires us to write and document test cases. We are quite automation-oriented and tend rather to generate tests dynamically: n test cases out of a rough 8,000! (that is, 8,000 factorial) possible test cases per day. So how may test cases do we have, according to how the standard counts them? 1? n? n × 1 sprint? 8,000 factorial? Are the executed tests the test cases or is the source code the test cases? Do we need to give our clients access to our automation Git repository? If so, how will that help them?

Also, we don't use a test plan. That has not stopped us shipping quality software to our customers for years, but since we can't map anything to our nonexistent document (the idea of a standard is that you can map things to it), that would mean if required to do so we would have to generate one. Worse than that, even the documentation we do produce (and we do) might not fit the model defined by the standard. Sometimes real, useful documents don't fall neatly into a predefined category. So we may have to create something that is not natural to our organization twice. Once, in the way we need it, and once, to fulfill a contract which we likely didn't review or approve.

Maybe the standard says we must create documents to create and maintain traceability. But most people don't read documentation. If you buy into the Agile principle that writing less documentation is better than writing more you will understand this point all too well. If you disagree, and feel more comfortable having more documentation, you may not. So such a standard by its nature would tend to make software development less Agile. Is my imagination running away with me? Would anyone really include a requirement to comply with a generic standard as part of a contract? Well, Dr Stuart Reid, convener of ISO/IEC JTC1/SC7 Working Group 26 which created the standard, has said "Imagine an industry where qualifications are based on accepted standards, required services are specified in contracts that reference these same standards, and best industry practices are based on the foundation of an agreed body of knowledge – this could easily be the testing industry of the near future." (http://www.testing-solutions.com/services/stqa/iso-29119-implementation?sid=1307 ; NOTE: requires email) So it sure sounds like that is the plan.

In other videos (including http://youtube.com/watch?v=e5fdJet4jpY retrieved 28th August 2014 1700 UTC), Dr Reid states that he does not think 29119 will change anyone's activities, unless of course their client requires it. He also notes that there are no best practices, but WG26 thinks the standard defines good practices. But if they are only good practices, how can they be standards? Is it a good practice to enforce, for example, email address conformity or are there standards? Whatever they are, testing standards tell you how to implement your testing rather than allowing that to be the polymorphic process it really is.

Part of the argument in favour of tester certification (I know I'm changing subjects but please bear with me) is that most people don't know what a good tester looks like, so need a method of identifying them. In the same way, most people who don't specialize in testing don't know the difference between good and best testing practices. They may see some standard mandated in a contract and not realize that will require double the effort for our team to deliver testing. Perhaps we can explain to the sales team why it is imperative they do not sign any contract mandating 29119, but that does not mean we can do the same to customers who do not know what good or bad testing looks like so cannot judge, other than by trusting in the standard to guarantee good testing.

To understand the true flavour of this debate, I strongly recommend listening to James Christie's talk on it at CAST 2014 (https://youtube.com/watch?v=A721ltyVw3o retrieved 28th August 2014 1700 UTC). The dichotomy is caused at least in part by personal taste and opinion. When personal taste and opinion starts to be enforced or coerced, people will start fighting. Many people see 29119 as a pathway to that, so they are fighting it.

Most opponents of 29119 don't want to burn anyone's work. Most of us just don't want it to be thrust upon us against our will, and vehemently object to the claim that this has been agreed upon by the entire international community.

– JCD, http://about98percentdone.blogspot.com/

All 'letters to the editor' received at editor@professionaltester.com are published

Not so big

by Reshama Joshi

Even if test volume is large, defects are simple

Reshama Joshi shares details of real project experience important to theory

The term "big data" is sometimes used quite generally, to mean the collection and management of data sets which are difficult to process using popular database management tools or traditional data processing applications because of their extreme size, diversity of source and format, and, resultantly of both, complexity. Besides that, there is often also unstructured data which cannot be handled by relational or object-relational DBMSs at all. Wayne Yaddow described the technical challenge in his tutorial on testing in data warehousing projects in the February 2014 issue of PT. He mentions in passing one of the more specific meanings of the term: using the data to derive business intelligence.

Testing a big data implementation intended to do that amounts to assuring the business intelligence is accurate. That requires first assuring that the data itself remains accurate in all important respects, then that the analysis algorithms are applied and their results reported accurately. This article describes a recent testing project which did both.

The application (see figure 1), to which I will refer by the fictitious name 'SpringBox', is owned by a global car manufacturer. It gathers data from social networking applications, directly and via company-specific RSS feeds. Its output is intelligence about what customers and the public are saying about the brand. The marketers who use that intelligence call the various reports they require by terms such as such as opinion mining, sentiment analysis, and predictive analysis and use them to perform, for example, ad targeting, search quality optimization and customer churn prevention.

Master test strategy
We were tasked to deliver end-to-end testing, so were obliged to apply testing at the interface of the connector APIs with Hadoop and at the interface between the data processing and analysis engine and the UI. We tested the functionality of these interfaces, and of the data processing and analysis engine itself, by considering them as separate black boxes.

Test strategy for the connector APIs (data fetching)
The input is the HDFS flat files into which data fetched from the APIs provided by the social networking applications and RSS feeds is dumped, and the output is the MongoDB collections created by MapReduce (including Hive and Pig) from them.

Before automating comparison of these, we established, by manual analytics inspection, that the input was valid. This was achieved, with the help of business subject matter experts and data architects, by studying historical sources (API responses and RSS feeds) and asserting how they should be manifested in the flat files.

Then, with the help of developers, we built a comparison utility. This required us to learn about NoSQL database types. Once built and tested, the utility enabled us to execute tests with high volumes of input data, increasing confidence that testing was sufficiently thorough.

Test strategy for the data processing and analysis engine
The input is the MongoDB collections and the output is ETLd (extracted, transformed and loaded) data stored on Oracle servers. The functionality is performed using the distributed ETL tool Scoop.

We tried again to build a comparison utility, but it became apparent before automation that it was not necessary because the ETL functionality and its output was quite simple. So we concentrated instead on the source and target data within MongoDB, performing mainly manual (aided by standard utilities and our own macros) testing based on our clear understanding of the data structures and data architecture involved, with respect to the data flow.

Test strategy for the UI
The input is the output of the (very simple) data emitter engine. The output, for compatibility with devices used to view it, is an Adobe PDF. We

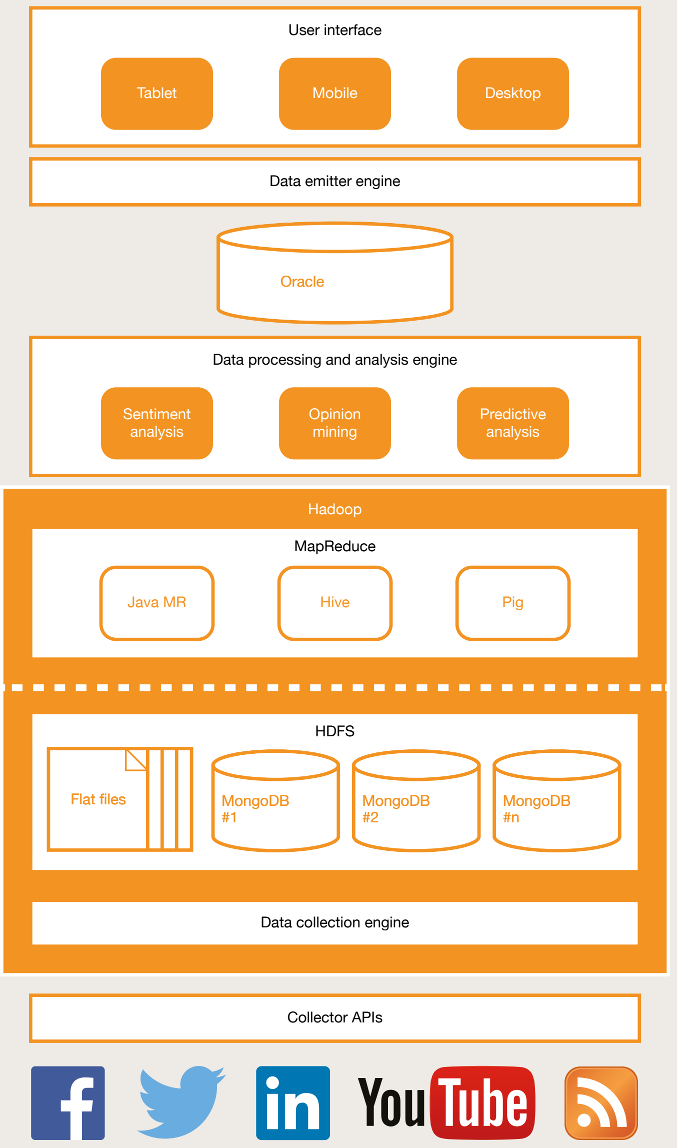


Figure 1: SpringBox layers

hoped a simple comparison of the two would again suffice. As readers who have had to deal with testing PDFs will already have guessed, making that comparison proved the most difficult task of the project. But it was just as well, because the attention we paid to it led to the detection of unexpected defects elsewhere. Sometimes the output, as would be presented to users, simply didn't make sense. We raised incidents we had not expected to need to raise, and their investigation showed their cause to be the quality of data at source. In other words: the code *invoking* the social media APIs, testing of which was not in our original test strategy, was defective. The defects proved simple to fix, so retesting, that is re-execution of our demonstrably good tests, at all three levels, provided a high level of confidence that the tested build of SpringBox almost always outputted accurate business intelligence ■

Reshama Joshi is global operations head of testing service line at L&T Infotech. She thanks her colleague Palak Kedia, delivery and practice head of data centric testing, who contributed valuable technical material to this article

Really risky

by Stefan Patry

Continuous risk-based test management

Stefan Patry
on preventing
test introversion

When you need to create a test plan, what is your first step? Start filling in a template based on a test documentation standard? Adapt an existing test plan, keep the generic stuff and replace the project-specific stuff? Gather as much information as possible, compile and refine it, and then hold an initial review to identify what is missing, wrong or noncompliant?

Any of these, done carefully, can lead to a result which can be verified as a good example of its type, and validated as having covered everything in its sources. But will it be effective, and how can we know whether or not it is? In other words, it should describe our test strategy and approach very well, but that does not mean they are right, or even good, for the software project or product.

I think most testers, and stakeholders, would say the answer lies in risk. The test plan should identify and quantify all risks and explain how and when they will be mitigated. Then its effectiveness can be measured by how closely to the plan that mitigation proceeds, and at what cost.

The trouble is the moment the plan is published it starts to become out of date. WIth every step of development and testing work, the likelihood and severity of identified risks changes, and new risks are identified. As all test planning standards acknowledge, the plan needs to be changed frequently. Managing that, and ensuring everyone is up to date with the latest version, has always been hard. In today's software organizations that embrace continuous, radical change even to requirements, very frequent releases and multi-project working by disparate teams, doing so with a manual, static document is impossible.

So the test plan needs to be automated and dynamic: stored and maintained using a tool which updates it automatically in real time according to information received from testing and development activity as well as manual intervention and communicates the result – the current plan – back to all those activities in ways they can use instantly.

But now that plan is based on requirements, test cases, development milestones and incidents – no longer on risk. The testing effort becomes introverted, basing its decisions on and measuring its progress against itself rather than the external, objective, real-world needs of the business. Error, waste and delay ensues. Furthermore, the plan is no longer comparable with that defined by any standard.

RBT at the heart of automated test management

At test management and consultancy provider *the test leaders* we believe strongly in risk-based testing and so are very concerned about this dilemma. Thinking about it and experiencing it so many times has led us, we believe, to the solution. After great success using it in our work with clients, we have recently made it available commercially as the first online risk-based automated test management platform: NoRizzk.com.

NoRizzk.com is more than a tool: it is an expert system. You don't start planning from scratch, or from another plan, but by choosing from comprehensive pre-defined risk categories, test policies, types, approaches and tools, entry and exit criteria and much more, composing the optimum plan for your specific project in an easy and standardized way.

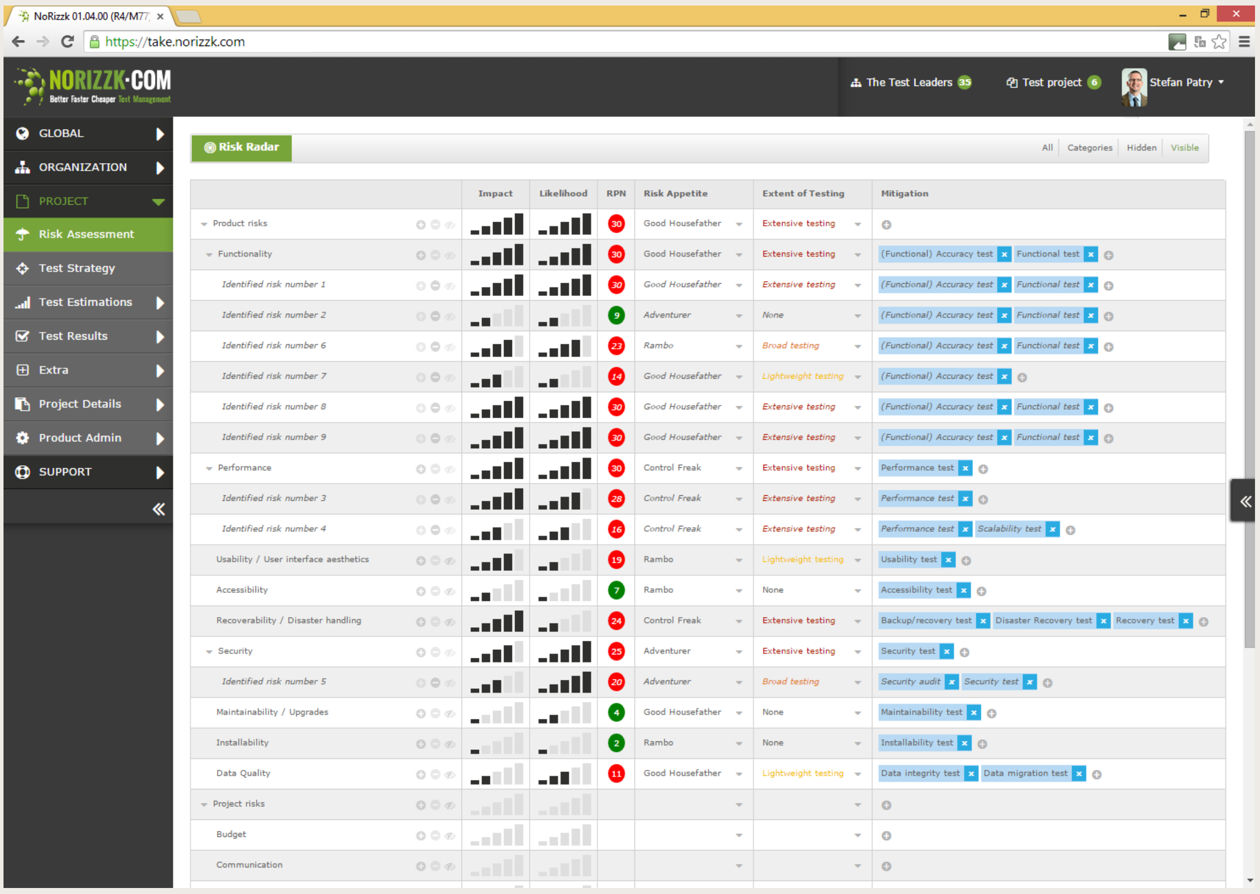
In the same way, progress against the plan is simple to measure and monitor and project control is achieved, as it should be, by continually refining the plan according to your strategy which remains focussed on what matters: risk mitigation.

At the click of a button at any time, the current plan can be output as a static, IEEE 829/ISO 29119-like, document.

RBT methodology in practice

NoRizzk.com implements the familiar risk-based testing approach – risk identification, assessment, mitigation and management – as explained in many testing books, standards and syllabuses, prioritizing both project and product risks according to impact and likelihood (see figure 1).

But that is not enough! in the real world, test managers need to take many more factors into account, including some which are not easily defined or quantified. NoRizzk.com goes beyond theory to deal with this by determining a detailed risk appetite profile. Again you are not asked to estimate numbers out of thin



	Impact	Likelihood	RPN	Risk Appetite	Extent of Testing	Mitigation
Product risks			30	Good Housefather	Extensive testing	
Functionality			30	Good Housefather	Extensive testing	(Functional) Accuracy test; Functional test
Identified risk number 1			30	*Good Housefather*	*Extensive testing*	*(Functional) Accuracy test; Functional test*
Identified risk number 2			9	*Adventurer*	*None*	*(Functional) Accuracy test; Functional test*
Identified risk number 6			23	*Rambo*	*Broad testing*	*(Functional) Accuracy test; Functional test*
Identified risk number 7			14	*Good Housefather*	*Lightweight testing*	*(Functional) Accuracy test*
Identified risk number 8			30	*Good Housefather*	*Extensive testing*	*(Functional) Accuracy test; Functional test*
Identified risk number 9			30	*Good Housefather*	*Extensive testing*	*(Functional) Accuracy test; Functional test*
Performance			30	Control Freak	Extensive testing	Performance test
Identified risk number 3			28	*Control Freak*	*Extensive testing*	*Performance test*
Identified risk number 4			16	*Control Freak*	*Extensive testing*	*Performance test; Scalability test*
Usability / User interface aesthetics			19	Rambo	Lightweight testing	Usability test
Accessibility			7	Rambo	None	Accessibility test
Recoverability / Disaster handling			24	Control Freak	Extensive testing	Backup/recovery test; Disaster Recovery test; Recovery test
Security			25	Adventurer	Extensive testing	Security test
Identified risk number 5			20	*Adventurer*	*Broad testing*	*Security audit; Security test*
Maintainability / Upgrades			4	Good Housefather	None	Maintainability test
Installability			2	Rambo	None	Installability test
Data Quality			11	Good Housefather	Lightweight testing	Data integrity test; Data migration test
Project risks						
Budget						
Communication						

Figure 1: risk assessment and prioritization

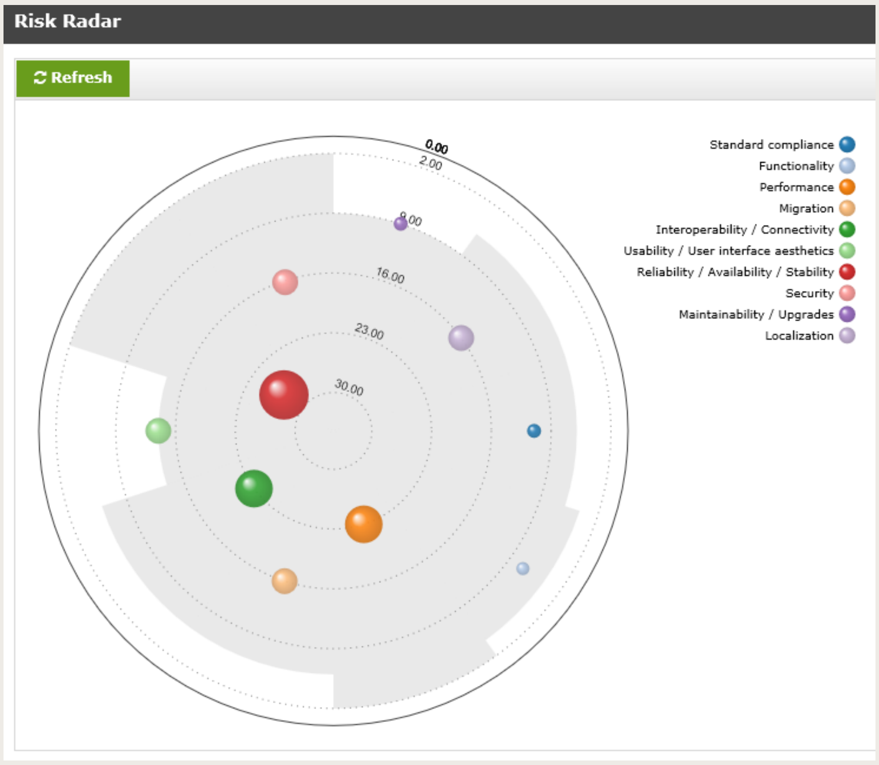
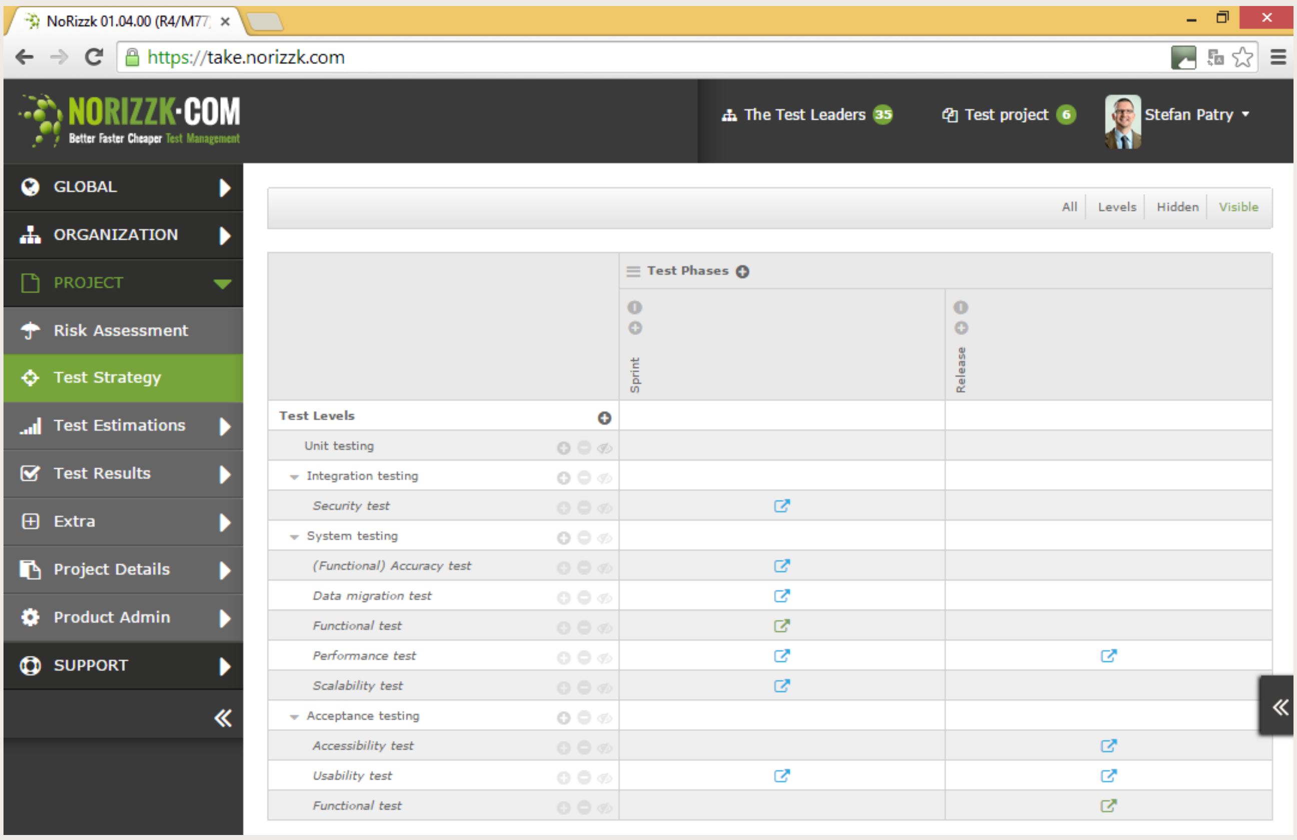


Figure 2: risk situation vs risk appetite



Test Levels	Sprint	Release
Unit testing		
Integration testing		
Security test	↗	
System testing		
(Functional) Accuracy test	↗	
Data migration test	↗	
Functional test	↗	
Performance test	↗	↗
Scalability test	↗	
Acceptance testing		
Accessibility test		↗
Usability test	↗	↗
Functional test		↗

Figure 3: test strategy matrix showing test level, type, phase and instance

air, but guided by an expert system offering predefined choices, per risk, that you combine and refine to capture exactly your current risk management strategy.

Decisions on test extent, approach etc are suggested based on the difference between the risk appetite and the current risk mitigation level, not simply on the risk prioritization. You control testing based on the risk situation *now,* not when initial planning was done. That situation is depicted graphically in NoRizzk.com's *Risk Radar* display (figure 2). Each coloured sphere represents a risk: its size, impact; its proximity to the centre, likelihood. The grey zone represents risk appetite.

NoRizzk.com also implements a standard, familiar test entity and process model (for example, it offers full integration with HP ALM), defining test levels, types, phases (including non-sequential phases such as sprints) and instances, all fully customizable to align with organizational and project test policy. These entities are displayed in the test strategy matrix (figure 3) which translates the current risk management situation directly to test activities, both analytical and empirical.

NoRizzk.com does not try to change the way you test, or manage testing: it just helps you to do it right. Right means always based on risk ■

Stefan Patry is CEO and managing partner of the test leaders (http://thetestleaders.com). He thanks the whole NoRizzk.com team for their valuable assistance with this article. A free trial is available at http://NoRizzk.com

---

NORIZZK·COM

Better Faster Cheaper Test Management

Want to save 20% on your testing budget?

Keep full control over your risks
Increase your test strategy efficiency
Maximize your product quality

Make your testing better, faster and cheaper...
...in Classic and Agile projects

The world's first risk-based test management platform
Get your free trial today at http://trial.norizzk.com

Help the afflicted

by Sakis Ladopoulos

How to fight for your right to test

Negotiation advice from PT's master test manager Sakis Ladopoulos

There is an old saw about management: that the three most important success factors are communication, communication and communication. It is a half truth that proves half truths can be more misleading than untruths and can cause more human pain.

Communication refers to the ways and means by which information is transferred. While obviously important, they are not more important than the content of the information which has value only if it is fit for its specific purpose. If it is not, communicating it contributes to failure not success.

There is an alternative interpretation, communication by an individual, which I will ignore here on the grounds that a manager who cannot express adequately what is in his or her mind is (or at least should be) a self-cancelling problem.

So what is the purpose of the information communicated? In a work environment it is to convey data and/or ideas, intending to support management policy or strategy. Discussion for its own sake is not only unproductive, but harmful, and should be eliminated.

When you go into a physical meeting, you have (I hope) specific aims related to creating, starting, expanding or maintaining business, and a strategy, even if rudimentary, to help achieve those aims. If not you, and probably the whole meeting, are doomed, because of you. Yet in corporate communication by any other means this is often forgotten and much vaguer aims such as the quest for absolute truths or the meaning of life, or more often simply to show off or win arguments on any subject however trivial, arise and distract us.

In business communication, rhetoric is anathema and logic is inadequate. To support any position or advance any strategy, data is required: preferably statistical, but at least metrical (ie involving measurement) and indicative of performance. Furthermore, *only data which actually does achieve one or both of those aims should be presented.* Some will say this is bad science, and in other contexts they would be right, but here we are not conducting science but business, and in business only one thing matters: sales. I don't mean by this that, for example, quality of delivery, compliance and ethics do not matter: rather that they do, because doing them well promotes sales and doing them poorly will always lead, even if not immediately, to loss of sales, or of the benefit of sales already made.

What has sales got to do with testing?
Nothing. Neither should testing have anything to do with sales.

Engineers often find understanding the business perspective on their work difficult and discouraging because it is aligned with their own perspective only by chance. Indeed, there are cases where it seems that business perspective diminishes the value of engineering work. As with testers and programmers, it's a mindset thing. To be effective, engineers need to act as though they believe that their job is to handle the very essence of the work at hand and that they know the deterministic truth about it.

Even if that were so (which it is not but that's a different article, maybe for *Professional Psychologist*) it is still those responsible for business decisions who should make them. Testers can label an incident critical or minor. They cannot say the critical one should be fixed first, because that depends on other factors which are not their concern: and should not be their concern, because the mindset testers, correctly, endeavour to stay in prevents them from considering those factors correctly. The converse – that those deliberately cultivating a business mindset should stay away from objective, isolated, low-level testing decisions – is obviously also true. Accepting these boundaries is essential to successful negotiation by testers with business.

How can testing sell itself?
In business-to-business negotiations the importance of understanding the other side's point of view is frequently emphasized. I have just argued that in the particular situation where testing negotiates with business for the mandate and resources to test, this is not the right way. It is very hard indeed for a tester to think like a businessperson or vice-versa, most if they try will not be able to do so adequately, and to the extent they do succeed they will diminish their ability to do their own job.

So I urge testers to continue to think like testers and do what testers do: and one of the ways that can be summed up, I think, is that testers provide answers to questions. Now answering questions can take a long time, if one waits for the question to be asked before starting to seek its answer. So testers also aim to predict the question in advance and have the right answer, supported by evidence, ready. That is what testers should do when entering negotiations with the business for which they work (or want to work). This strategy can also be compared with the most popular technique used by job interview candidates.

Dealing with hostility
Previous contributors to PT have discussed at length the apparent contradiction that testers must do all they can to detect defects but must not want defects to exist. The fact that some misguided testers want defects to exist so they can detect them is not their fault but that of their misguided masters who do not understand testing or how to measure its effectiveness.

Its consequence is that many in IT, including some at high corporate levels, think testers are sick. This belief often gives rise in negotiation to hostile questions, asked with sincerity but rooted in ignorance. For example: why do we need you to define phase exit criteria just to hold up production by refusing to sign them off? What good does it do if you antagonize and demoralize the developers by analyzing and reporting their mistakes in gory detail? How can we keep the confidence of our clients if you wash our dirty linen in front of them? Where is the return on investment in this test tool which will help you detect more defects that don't matter? Why do we need to do something which, if it does not cause change to the product, is useless; but if it does is therefore dangerous? Why do you request money to bring chaos?

Some reading this will think some or all of these questions unthinkable, Others, less lucky, will recognize their theme and the attitude behind it as something they deal with often if not continually.

Testing as the bearer of bad news
History relates many messengercides. The trouble with testers, from the point of view of everyone else, is that testers tell the truth and everyone else, sometimes, would rather that truth were not told.

According to Herodotus, Xerxes commanded that the sea be whipped because it had washed away his new bridge. Xerxes was probably mad but you wouldn't do business with him if you had the nerve to tell him so. You probably wouldn't do anything ever again. Again it is not a matter of right or wrong. Trying to explain to him, a man who really wants a good bridge and is interested in bridge building, how to make the bridge better might give you more chance. But, if you won his trust and then later he found out you had being lying, a bad fate would befall not only you, the worm who deserved it, but honest, hard-working innocents trying to find out the truth to tell him.

Don't let this be you. Never say anything like "we won't raise incidents that we, or you, consider unimportant" or "we'll talk with developers about incidents before we raise them". That is trying to sell him something he should buy, but does not understand why, by telling him the reason he should buy it is that it is no good. He may be mad but he is not stupid. Even he knows not to trust a liar.

The best strategy is to promise to answer his questions truthfully and dare him to listen. Tell him: we are not the good guys who will tell you what you want to hear; we are the bad guys who will tell you the truth. But we are *your* bad guys and will work our hardest to make sure that what we tell you (and only you) really is the truth.

Answering how, what, and when questions
Once you have sold that concept, some obvious questions will come to the mind of the prospect: how much testing is

needed? How many testers? Why do we need to test a particular release of which the developers say they are confident? What makes a fix need a retest? When do we need to regression test, and to what extent? What methods and tools should be used?

These are testing questions, so answer, politely, along a line similar to "the only way to know that is to do a small amount of initial testing. Please will you sanction that, and then I'll give you the definitive, explicit answer(s) before this time next week".

Be prepared also, if asked, to hazard a guess immediately. How to guess is a personal decision: you could make the best honest guess you can, throwing yourself on the mercy of fate; you could try to guess high, that is expensive, choosing a high-risk, high-payoff gamble; or you could try to guess low, improving your chances of living to fight another day. Personally I favour the latter: after all, your low guess may prove to be approximately right, and if not you may gain at least an opportunity to present a better guess with justification.

Answering why questions

More likely, and better, business questions are concerned with seeking that justification: for example "why do we need to test this release?", "why should we adopt this standard?", "why should we buy this tool?", "why should we trust you to answer these questions?". These and similar questions are instances or encryptions of the fundamental one: "why should we pay for testing?".

Make sure your prepared answer relates directly to increasing business sales. In the context of the product or and how it is used, point out that quality leads, repeat sales and growing market share all come about *only* because of consistently successful deployment and use of the product: that is, timely release of good quality software, regardless of whether the offering is the software itself, or something else that depends upon the software.

Cashflow is king: risk is boring

Be positive. Testing negotiations – and testing product and service marketing efforts – are very prone to the mistake of going on about the negative, what will happen if you don't test well.

It is a mistake because successful business people are almost always confident. Threatening them with consequences of failure, project or product, even with real examples, will not work because they will not believe it will happen to them but do believe in their ability to manage it away if it does.

They also believe very strongly – and rightly so – in the real danger of the business failing due to spending too much against too little revenue.

So do not say "doing this testing well will reduce risk". Risk is of the future and saying that risk can be or has been reduced is speculative. You may strongly suspect it, but you also know you are not certain. Anyone, to sell something honestly, must believe in it.

Rather say "doing this testing well will improve the product faster so make it sell better, make its development and maintenance cheaper, and get new versions of it to market earlier". Say that knowing, as a tester representing all testers, it to be true ■

Frequent PT contributor Sakis Ladopoulos (theodosios.ladopoulos@hotmail.com) is a test manager at INTRASOFT International and an independent QA and test management consultant

OCCYGEN

Manage your test cases easily

Sign up and get your package

www.occygen.com

Automate processes

Different packages at your disposal

Download

Synchronise your data with other tools

Bugzilla

JIRA

mantis BUG TRACKER

Support team available

support@occygen.com

"THE BITTERNESS OF POOR QUALITY REMAINS LONG AFTER SWEETNESS OF LOW PRICE IS FORGOTTEN"
BENJAMIN FRANKLIN

"Good enough" is just not good enough when it is about a product's quality...
..and we will not let you settle for less either!

Our services include...

- Functional Testing
- Performance Testing
- Automation Testing
- Cross Platform Testing
- Penetration Testing
- Code Reviews
- Mobile Application Testing

Kualitatem
QUALITY GROWTH PARTNERS

WEB: WWW.KUALITATEM.COM
EMAIL: INFO@KUALITATEM.COM
PHONE #: +1 415 800 4705

DUBLIN 2014

EuroSTAR
Software Testing Conference

JOIN US AT **EUROPE'S LARGEST**
SOFTWARE TESTING
CONFERENCE & EXHIBITION

60+ Sessions Including 6 Keynotes • 5 Full-Day Tutorials
6 Half-Day Tutorials • 40 Track Sessions • 3 Workshops

+ An Amazing Social & Networking Programme

VIEW FULL PROGRAMME HERE

EuroSTAR
Software Testing Conference | Dublin 2014
Nov 24-27

VISIT **www.eurostarconferences.com** OR
CONTACT **info@eurostarconferences.com** FOR MORE INFO