QUALE

Quale 3/2014

The testers and coding debate: Can we move on now?

Paul Gerrard

Competitive Planning Chapter 1 - Objectives

Tom Gilb

Test automation patterns

Seretta Gamba
Dorothy Graham

TestingCup 2015

Software

QS-TAG 2014

Test Organisation State of the Art

5 Tracks to the power of 10 Slots =
9.765.625 combination options
for your very own Software-QS-Tag

Register now!
www.qs-tag.de

6 th & 7 th November 2014 in Nuremberg

© Dmitry Vereshchagin - Fotolia.com

Supported by:

German Testing Board
www.german-testing-board.info

Powered by: iX

Exhibitors:

dpunkt.verlag
FINARIS FINANCIAL SOFTWARE PARTNER
InterFace AG the face of informatics
imbus
Ranorex
exept SOFTWARE AG
hp
Softwareforen Leipzig

www.qs-tag.de/english

imbus AG, Kleinseebacher Str. 9, 91096 Möhrendorf, GERMANY,
Phone +49 9131 7518-0, Fax +49 9131 7518-50, qs-tag@imbus.de

Content

DIFFERENT POINT OF VIEW

5. Music in Testing: Top of the Pops
Bartłomiej Prędki

7. Music in Testing: Let there be Rock!
Krzysztof Chytła

SOFTWARE TESTING

9. Test automation patterns
Seretta Gamba, Dorothy Graham

18. The testers and coding debate: Can we move on now?
Paul Gerrard

SOFTWARE ENGINEERING

25. Competitive Planning. Chapter 1 - Objectives
Tom Gilb

40. An overview of the SQuBOK® - Software Quality Body of Knowledge - and its benefits in the context of global collaborations for software quality
Susumu Sasabe

EDITORSHIP

Chief editor
Karolina Zmitrowicz
karolina.zmitrowicz@quale.pl

Deputy chief editor
Krzysztof Chytla
krzysztof.chytla@quale.pl

Editors
Bartłomiej Prędki
bartlomiej.predki@quale.pl
Michał Figarski
michal.figarski@quale.pl

Cooperation:
Tomasz Olszewski
tomasz.olszewski@quale.pl

Website:
www.qualemagazine.com (ENG)
www.quale.pl (PL)

Facebook:
http://www.facebook.com/qualemagazine

SEETEST
2014 SOUTH EAST EUROPEAN
SOFTWARE TESTING CONFERENCE BUCHAREST

25-26 September
BUCHAREST, ROMANIA



Speakers:

- Rex Black, USA
- Erik van Veenendaal, Holland
- Graham Bath, Germany
- Geoff Thompson, UK
- Yaron Tsubery, Israel
- Vipul Kocher, India

20% EARLY BIRD DISCOUNT!
1 - 31 AUGUST
REGISTER NOW!

www.seetest.org **info@seetest.org**

SEETB
South East European Testing Board

ANIS
employers' association of the software and services industry

QUALITY HOUSE

Bartłomiej Pręski

Music in Testing: Top of the Pops

"Stop, summertime!" [MC Hammer – U can't touch this]. Welcome my Dear Fellow. As you know, summer is the time of a beautiful weather, relaxing sound of waves, mojito and naked wom… well… I mean… and mojito. It is an obvious fact that one of the products – except for drinks – that are good sellers during summer is music – specifically one of its genres. Of course, there's nothing that fits the atmosphere of the summer better than Pop.

Recently, sitting on one of the beaches somewhere in the southern Europe, I came to the conclusion that, in principle, our profession perfectly resonates with the vibes of that music. Basically, when getting up in the morning with the perspective of the immemorial struggle with developers, analysts and management, the only thing you could say is "sometimes I feel I'm gonna break down and cry" [Freddie Mercury – Living on My Own]. And the leitmotiv of the day is Staying Alive [Bee Gees].

In fact – as you remember we've been recently discussing about joy and torment of testing – it's not that bad. Huge part of work-related communication can be done with use of Pop music. For example, after finding a critical bug, we can call the Dev Lead singing, "I just call to say…"[Stevie Wonder - I just call to say I love you]. No one deserves our love more than developers creating crappy code – thanks to them we have our jobs! Notwithstanding, it would be better of us when the person on the other side of the phone didn't ask, how much we loved them – we would be simply forced to answer "truly, madly, deeply" [Savage Garden - Truly Madly Deeply] with our fingers crossed behind our back.

It's getting much worse when we have to leave your comfort zone and go to developers in person – the world doesn't seem to be too perfect when we enter their section muttering "As I walk through the valley of the shadow of death…" [Coolio – Gangsta's Paradise]. Remeber, you can easily knock them off balance by saying with a hushed voice „Every fix you make, I'll be watching you" [The Police – Every Breath You take]. And just to make your message stronger, you can show them the famous eyes-fin-

gers gesture, just like Robert De Niro in Meet the Fockers. Oh, wait, it should be Pop, not Hollywood…

One of my favourite areas of testing is requirements analysis. It is widely known, that - in a perfect world – the testing process should begin in a very early phase of the project. When poring over documentation, I would often like to ask the author "Tell me why…". Moreover, this phase can be heavily affected by project managers, who force their last-minute corrections in : "Yes, I know, it's too late, but I want it that way" [BackStreet Boys - I Want It That Way].

Now, let me wrap up the testing process. We all know, that at the end of the project, it's required to prepare The Testing Report. In my mind I see the faces of the project manager, the customer and management representatives who read about the terrible quality of the product. Yes, those are the same guys who stinted money on testing. The further they wade in the report exposing their incompetence, the more they want to say "Don't tell me 'cause it hurts" [No Doubt – Don't Speak]. Some people like to learn things the hard way.

Of course, apart from the purely testing activities, there are some brighter parts of our profession, especially, when we take part in recruitment events like job fairs. For example, when speaking with a female internship candidate who is so "young and sweet, only seventeen" [ABBA – Dancing Queen], you convince yourself that the world is not such a bad place, in fact, it has a pretty face too. And in the evening, having the sense of a job well done, we can return home, dancing like John Travolta in "Saturday Night Fever". And later, when looking in the mirror with an impish smile, you can sing to yourself like Rod Steward – Do Ya Think I'm Sexy? Let the exploratory testing begin!

As you see – my Dear Friend – Pop music is like testing and testing is like Pop music. Although, I'm deeply convinced that you have a different view on this topic…

P.S. Of course my favourite music band is Crash Test Dummies ;)

AUTHOR

Bartłomiej Prędki

I've started my professional experience in 2004 as a tester of mass-market mobile applications. Within next years I gained an experience in Testing and Quality Assurance areas, mostly focused on Telecommunications industry.

During my career I was involved in testing, managing testing processes, training, technical support, requirement analysis, recruitment, technical documentation creation and review. Besides my mobile and telecommunications experience, I was also involved in financial and banking systems related projects. Currently I possess the role of QA Team Lead.

I'm a holder of two ISTQB Advanced certificates: Technical Test Analyst and Test Manager

I live and work in Wroclaw, Poland.

Krzysztof Chyłła

Music in Testing: Let there be Rock!

Hallowed be thy name Bartek [Iron Maiden]! I'm both happy and concerned for you. Is everything all right? Mojito drinks, sandy beaches and women in négligé must have driven you crazy like Britney in her prime, but Pop? Come on man, give me a break! What happened to good old sex, drugs and – above all – rock'n'roll? What can I say… de gustibus non est disputandum.
Our profession is like Pop? Whoa, brother, that's preposterous. Did you come to this conclusion under the influence of something funny? Or is it lack of vaccination? Yeah, that certainly must have been the latter; or a mild sunstroke. Please let me help you realize how wrong you are. There is still hope. Read through and see the doctor immediately.

It's common knowledge that testing is like Rock. How come? That's simple. Here's an insider's look on the testing process: tune up here, turn down there, adjust parameters, fix the input and check the output. Rings any bells? Wire things up, use MacGyver tape, follow safety procedures, verify functional (vocal) section, validate regression (percussion) section, then launch - isn't it the test environment setup? I'm not sure about you, but I can definitely hear the buzz of the amps before the Test Lead starts his exploratory (guitar) solo. All is set and done, we're back together, on the road, it's time to fly [Manowar – Return of the Warlord]. Then suddenly, bam! A bug pops up, jumps like an electric spark from one component to another [Van Halen - Jump], and you know, that "you've been missing that one final screw" [Queen – I'm going slightly mad].

I'll go one step further down the valley of the shadow of death and say that testing is like Heavy Metal. Why? Business demands products to be tested, tested by the best. When the testing's over, all the metrics done, we were born to win – number one! [Manowar – Number One]. Heavy duty testing assures unrivaled quality.

I hope that you've understood your grave mistake by now, but wait, there's even more!.

Starting your day like a wimp? I could expect that from a junior apprentice at the Tower of Testing Arts but not from you, a battle-hardened veteran supposed to lead not whine. Unbelievable. Is there a guardian of the blind (at the cost of poor quality) still dwelling in you? Remember last time you had spoken the words of metrics? Precisely, "then there was silence" and awe [Blind Guardian]. Never forget "that flame, that burns inside of you, hear the secret harmonies" [Queen – A kind of magic]. That test – it is a kind of magic!

Now you're talking! Crack you fingers like and get down to the business whistling "die, die, die my darling" [The Misfits] to the system under test that's been waiting for you your whole holiday (a freelancer with flexible work ethics would probably add a jar of whisky to that test harness). Thou shall not be afraid of face-to-face confrontation too as you know "for whom the bell tolls" [Metallica] when the force of regression result is with you.

Analyzing requirements in the early stage of a project is truly like playing with madness [Iron Maiden – Can I play with madness] based on some crystal ball insights. They're like hell – never freeze. Sounds like a nightmare. So, who you gonna call [Ray Parker Jr - Ghostbusters]? A Bugbuster! A true tester will "face it with a grin, is never giving in" [Queen – The show must go on] even, if it's a one way ticket to hell [The Darkness – One way ticket to hell]!

Testing – my love – is like a rock far out in the sea with waves of false pretenses crushing into it [Masterplan – Love is a Rock]. Quality's been through worse, it shall prevail. Summarizing, if love is a rock and testing is (my) love, then there is nothing else left but to conclude that testing is like Rock \m/ my Friend.

PS.
Watch out for those dancing seventeens. Things might not be as you believe they are (unless you test). Imagine yourself caught in flagrante by her daddy screaming, "Hey?! What's this supposed to be? She's only sixteen!" [Manowar - Warlord].

AUTHOR

Krzysztof Chytla

Test manager, designer and automation specialist with wealth of experience in embedded systems domain. Participated in big international projects assuring the highest product quality. Flesh and blood tester curiously analyzing rapidly expanding world of new technologies.

Author of translations and publications. Wroclaw University of Technology, Faculty of Electronics graduate. Trainer and coach passionate about acquiring and sharing knowledge.

On a personal note big fan of fantasy, science fiction and board games accopanied by a a glass of single malt whisky - an editor's best friend.

Seretta Gamba
Dorothy Graham

Test Automation Patterns

Patterns and Test Automation

The "Mother of all Software Pattern Books" Design Patterns [DP] was published in 1995 by the so called Gang of Four (GoF). The book is more than a collection of design patterns for the development of object-oriented applications, it describes what patterns are for (how they help solve design problems) and how to select and use them. The book has had at least as much influence on software development as the Agile Manifesto [AM], that permanently changed the way software is written.

Many other books on patterns followed (see References), but they have been mainly written for software developers; even testing patterns have only been published for Unit-Tests. As of now (2014) testers have not profited from them.

For Test Automation Patterns it all began when Dorothy Graham asked Seretta Gamba to contribute a chapter to her new book "Experiences of Test Automation" [ETA]. When the book finally appeared (2012), Seretta was really curious to read what the other collaborators had written. Some of the best features in the book are the "Good Point", "Lesson" or "Tip" notes that the authors included to emphasize especially interesting topics. Already after reading the first few chapters Seretta noticed that some notes were coming up again and again. A bell rang: PATTERN! PATTERN! PATTERN!

Seretta was well acquainted with patterns since she worked at least half of her time as a developer, but when she had her test automation hat on, it never occurred to her to look for patterns. She used them unconsciously, just as apparently all the other contributors to the book also had done.

After realizing this, Seretta immediately searched the Internet for a book about test automation patterns…and found only ref-

erences to Unit-Test Automation, nothing else! She decided on the spot to write the missing book herself. She asked Dorothy Graham if she would be interested in doing it with her. Dorothy didn't completely turn her down at first, but promised to review it. After working for about 8 months, Seretta sent the first draft to Dorothy and the review was really enthusiastic. The collaboration between Dorothy and Seretta had begun and the original book has long since been transformed into a wiki.

Test Automation Patterns

In the context of test automation, a pattern can be either a description of how some testware has to be in order to solve some test automation problem, a rule about how to perform a particular step in a test automation process, or a suggestion about how to resolve a management issue. In other words a pattern is expert knowledge proven by repeated experiences. A pattern shows the way to help solve some test automation issue.

Patterns do not exist in a void: each is a solution to an issue that occurs under some particular conditions or context. Also patterns are often associated with other patterns either because they can only be implemented using other patterns or because they can only be applied after other patterns have been put into practice. As an example think about the pattern "car": can you really believe that we could use this "pattern" so successfully if we hadn't also implemented the patterns "paved road" and "gas station"?

The relationship to other patterns forms the "grammar" of a pattern language. Just as in English you cannot write the parts of a sentence in a gratuitous sequence, but you must follow some rules, so also with test automation patterns you have to follow their natural hierarchy.

Lastly a pattern is not:

- a finished solution that you can just "plug in" directly to your situation
- prescriptive (you must do this)
- a step-by-step procedure (do this first, then that)

We have classified the patterns into four categories:

- Process Patterns: how the test automation process should be set up or how it can be improved.
- Management Patterns: how to manage test automation both as an autonomous project or integrated in the development process
- Design Patterns: how to design the test automation testware so that it will be efficient and easy to maintain
- Execution Patterns: how to ensure that test execution is easy and reliable.

To be able to recognize one when we are talking about a pattern we write it in capital letters (PATTERN).

We describe for each pattern the contexts in which it can be applied, the actual recommendations and how to implement it, eventually suggesting other patterns. When known, we have added examples on how it has been applied. The mind map (Figure 1) gives an overview of the patterns we have collected as of July 2014.

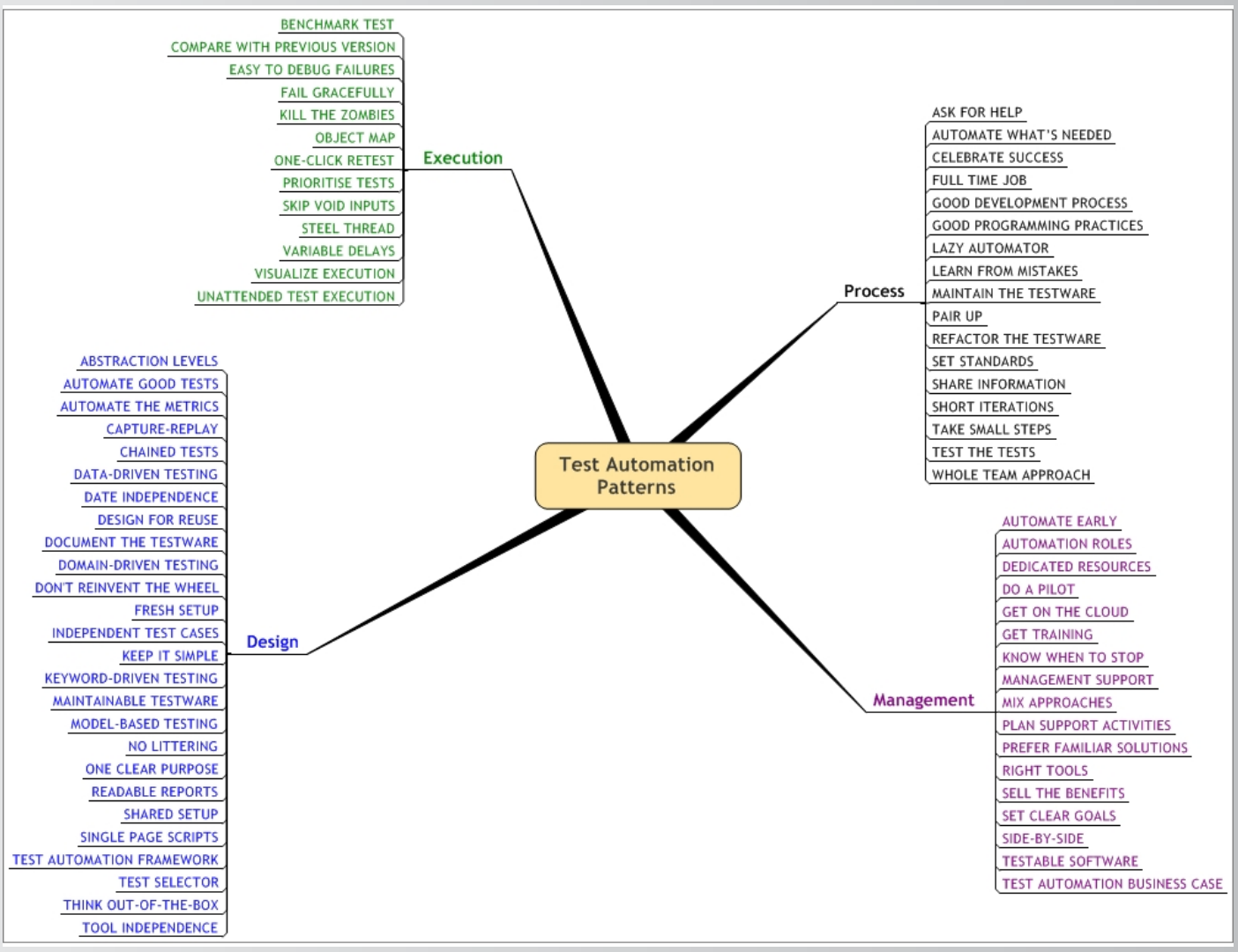

Fig. 1. *Test Automation Patterns*

Test Automation Issues

Since only a recognized problem can be solved efficiently, in order to be able to use the patterns as solutions we must first describe the issues that testers have to face when they tackle test automation. We have deliberately chosen to call any type of test automation problem an issue. This is because issues can not only be problems like high maintenance costs but also simply tasks that have to be done as when you start test automation from scratch or you have to select a new tool or team.

Test automation issues are manifold. Some are technical in nature, such as inefficient failure analysis or brittle scripts. However, one of the main reasons for failure is to concentrate exclusively on the technical aspects. Other issues are related to the way you work, such as late test case design, or when automation seems to get off to a good start, but then grinds to a halt. Others are management issues, such as high ROI expectations. Some issues may arise due to both technical and manage-

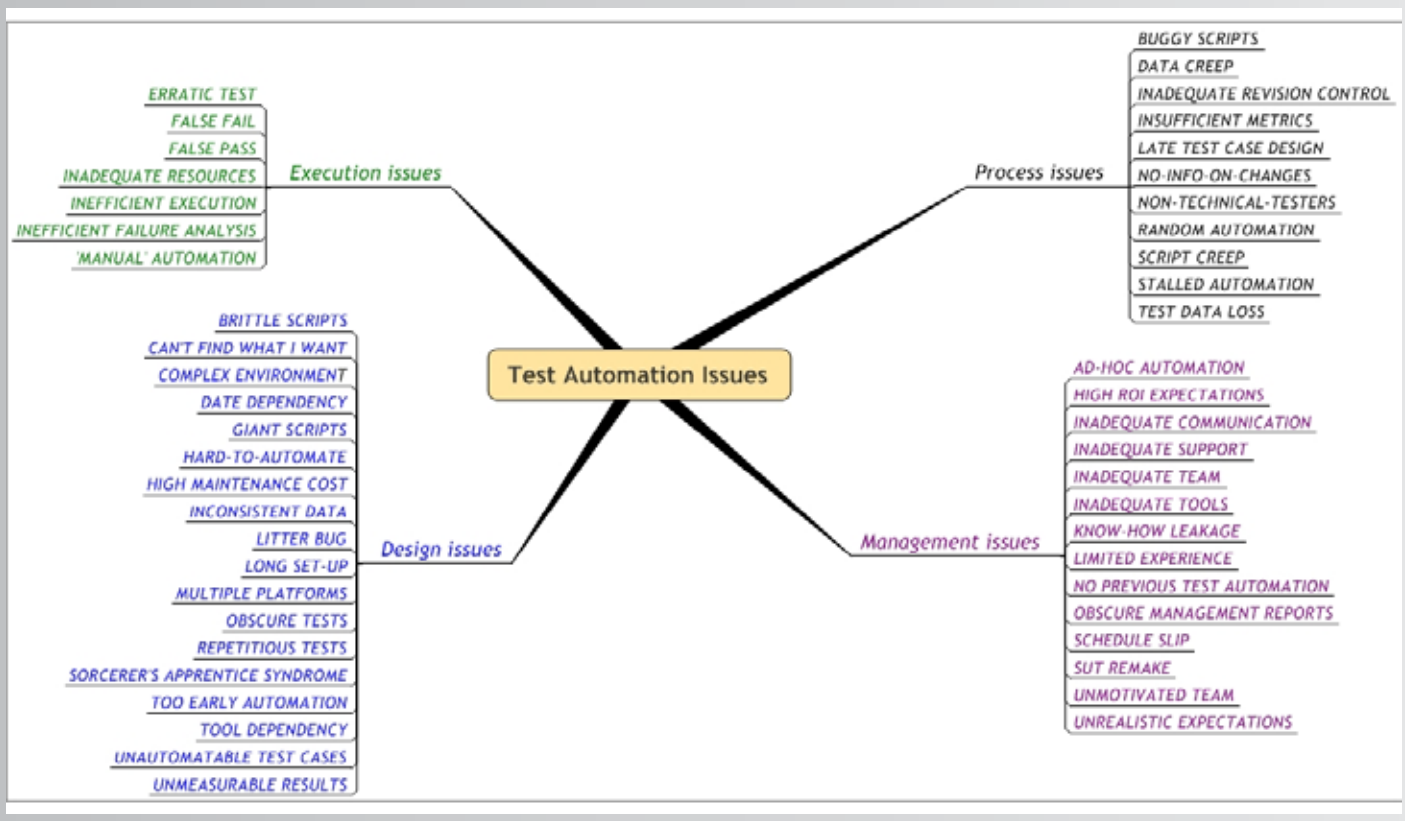

Fig. 2. *Test Automation Issues*

ment problems.

We have classified the issues into four categories:

- Process Issues: the way we work with automated tests and tools
- Management Issues: issues of management, staffing, objectives
- Design Issues: testware architecture, including maintainability
- Execution Issues: the running of tests in their automated form

To be able to recognize one when we are talking about an issue we write it in italic capital letters (*ISSUE*).

We describe every issue with examples in different contexts and then give suggestions as to which patterns to apply in order to solve it. The mind map (Figure 2) gives an overview of the issues we have collected as of July 2014.

How to work with issues and patterns

Let's see with a short example how issues and patterns work together. Let's start with an issue that many automators have had to tackle and show how it leads to the patterns that help solve it.

We will look at the issue *BRITTLE SCRIPTS* and the pattern MAINTAINABLE TESTWARE.

In the issue examples we list some possible occurrences of the issue (eventually we want to have examples in differ-

Issue Summary
Automation scripts have to be reworked for any small change to the Software Under Test (SUT)

Category
Design

Examples
Scripts are created using the capture functionality of an automation tool. If in the meantime something has been changed in the application, the tests will break unless recorded anew

Questions
How do you develop automation scripts?

Resolving Patterns
Most recommended:

- DATA-DRIVEN-TESTING: this is the pattern to implement if up to now you have only used capture / replay. The constant values captured by recording the tests are substituted with variables that are read from external files. The rework effort due to changes in the SUT will be reduced since the scripts can be reused for any number of tests.
- KEYWORD-DRIVEN-TESTING: This pattern involves more development effort than DATA-DRIVEN TESTING, but is much more efficient on the long run. Using keywords to drive the tests enables you to write test cases that are practically independent from the SUT. If the SUT changes, the functionality behind the keyword must be adapted, but most of the time the test cases themselves are still valid
- MAINTAINABLE TESTWARE: This is the pattern to apply if you want to get rid of the issue once and for all. If you haven't implemented it yet, you may want to apply at least some aspects of this pattern.
- MANAGEMENT SUPPORT: This is the pattern to apply if you are missing support or resources that you need in order to develop MAINTAINABLE TESTWARE

Other useful patterns:

- GOOD PROGRAMMING PRACTICES: This pattern should already be in use! If not, you should apply it for all new automation efforts. Apply it also every time you have to change current testware.

MAINTAINABLE TESTWARE

Pattern Summary
Design your testware so that it does not have to be updated for every little change in the Software under Test (SUT).

Category
Design

Context
This pattern is applicable when your automated tests will be around for a long time, and/or when there are frequent changes to the SUT.
This pattern is not applicable for one-off or disposable scripts

Description
Identify the most costly and/or most frequent maintenance changes, and design your automation to cope with those changes with the least effort. When adjustments really are necessary, then they should be relatively easy to implement. For example, if objects are frequently renamed, construct a translation table from the name you want to use in the tests, and put in whatever the name of the object is for the current release of the SUT (OBJECT MAP).

Implementation

Some suggestions:
- There are many options to make and keep the testware maintainable, but to adopt a GOOD DEVELOPMENT PROCESS and GOOD PROGRAMMING PRACTICES is a very good bet: what works for software developers works for test automation just as well!
  - For example, if your scripts are mainly „stand-alone" without much reuse, and automators are frequently re-inventing similar or even duplicated scripts or automated functions, then GOOD PROGRAMMING PRACTICES are needed, particularly DESIGN FOR REUSE and OBJECT MAP.
- Implement DOMAIN-DRIVEN TESTING: test automation works best as cooperation between testers and automation engineers. The testers know the SUT, but are not necessarily adept in the test automation tools. The automation engineers know their tools and scripts, but probably wouldn't know how to test the SUT. If the testers can develop a domain specific language for themselves to use to write the automated test cases, the automation engineers can implement the tool support for it. In this way they will each be doing exactly what they do best. The advantage for the automation engineers is that in this way the testers will maintain the tests themselves leaving the engineers more time to refine the automation regime.
- For example, if you have structured and reusable scripts, but there is a shortage of test automators to produce the automated tests (or they are short of time), this pattern gives the test-writing back to the testers, once the automators have constructed the infrastructure for the domain-based test construction. Other useful patterns are ABSTRACTION LEVELS and KEYWORD-DRIVEN TESTING.

Potential problems
Don't leave it too late to build maintainable testware - this is best thought of right at the beginning of an automation effort. (Although it is also never to late to begin improvements.)

Issues addressed by this pattern
- BRITTLE SCRIPTS
- DATA CREEP
- HIGH ROI EXPECTATIONS
- NO PREVIOUS TEST AUTOMATION
- OBSCURE TESTS
- SCRIPT CREEP

ent contexts). Some questions should also help you recognize if the issue is actually the one that you have to tackle.

Finally we recommend a number of patterns and give also hints to other useful ones.

Since the patterns are in alphabetical order it's not important which one you examine first. We have tried to give both the patterns and the issues sensible names so that you can kind of guess their contents just by reading the name.

For our example let's take a look at MAINTAINABLE TESTWARE.

For every pattern we give the context where it can be applied successfully or where it wouldn't be worth the trouble to invest in it.

In the description we outline what the pattern recommends and then give one or more suggestions about how to implement it. We also give a short list of problems that could arise when applying the pattern.

Finally you should be able to find the issue from which you started in the list of the issues addressed by the pattern.

When you decide that the pattern you are examining is not the one you need, you just go back to the original issue and examine the next one.

On the other hand, if more than one pattern would be useful, then just start implementing one. That done, you can get back to the other one(s).

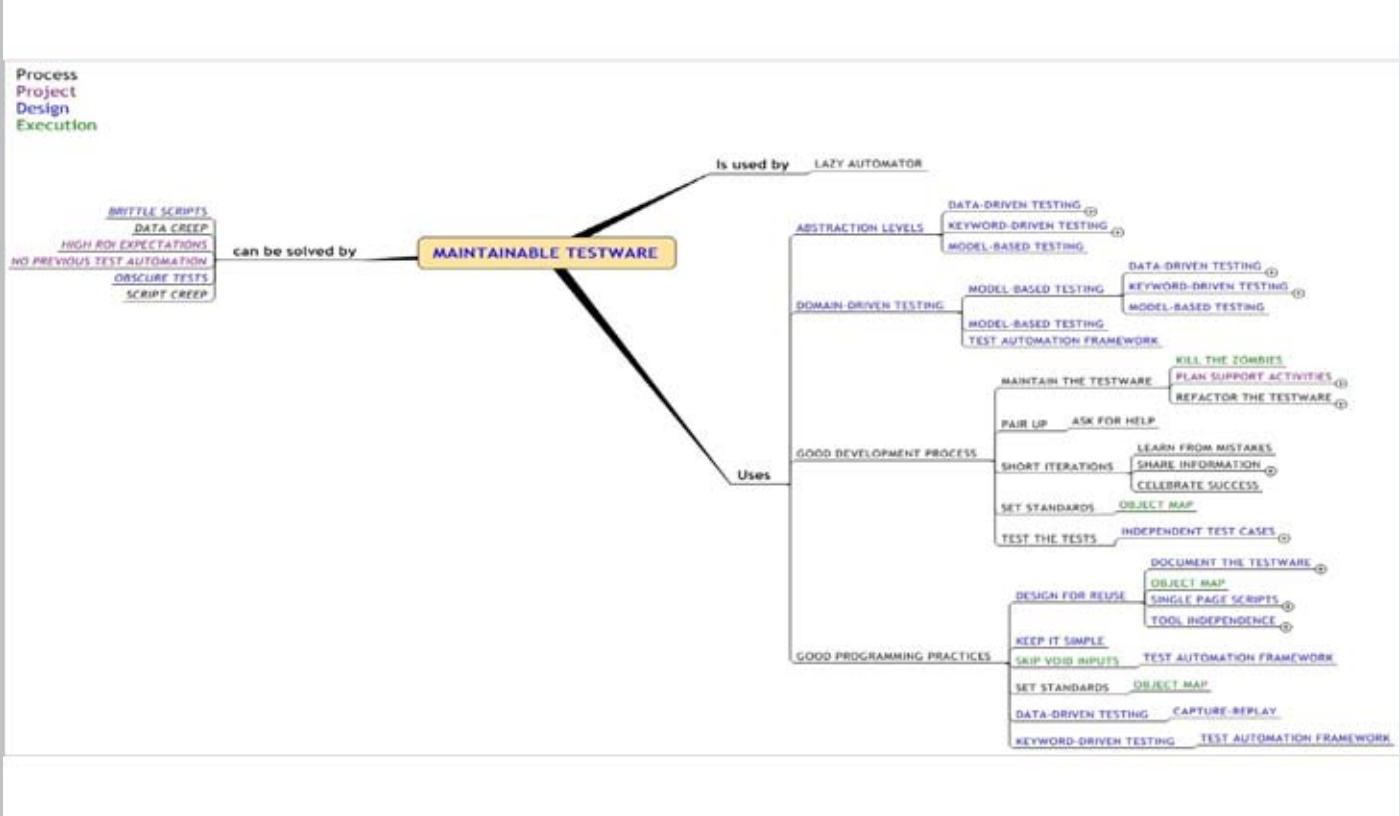

Fig. 3. *Maintainable Testware*

What is also important to notice here is that in order to implement this pattern you need to implement some other patterns first (i.e. OBJECT MAP, GOOD DEVELOPMENT PROCESS, GOOD PROGRAMMING PRACTICES, DESIGN FOR REUSE, DOMAIN-DRIVEN TESTING, ABSTRACTION LEVELS and KEYWORD-DRIVEN TESTING). Some of them will probably also need other patterns to be implemented before they themselves can be applied. It can get quite complex as you can see in the mind map (Figure 3).

Here again, how do you know where to start? Well, since the recommended patterns should all be implemented, you could start anywhere, but we suggest that you go first with the easiest ones, the ones where you will get improvements fast. Once you can show that your automation is getting better, it will be easier to convince people (especially management) to support you when tackling more complex tasks.

In this example you could start for instance with GOOD PROGRAMMING PRACTICES / SET STANDARDS. Note that SET STANDARDS is also recommended by GOOD DEVELOPMENT PROCESS so applying it would start you off for both patterns (GOOD PROGRAMMING PRACTICES and GOOD DEVELOPMENT PROCESS).

Conclusion

So what issues are giving you problems in your test automation? Would you like to have some ideas for how to address them? We hope that the issues and patterns that we are putting in our wiki will help you to do better system-level test automation.

Have you experienced one or more of these issues and/or patterns? If so, we would like to hear from you, or have you write up a few sentences about your experience inside the relevant issue or pattern. Viewing is open to all; to write to the wiki, just ask to be invited, and we will be happy to see your comments. We also have a discussion page for general comments, disagreements, etc.

TestAutomationPatterns.wikispaces.com.

AUTHORS

Seretta Gamba

Seretta Gamba has over 30 years' experience in software development. As test manager at Steria Mummert ISS GmbH she was charged in 2001 with the improvement of the test automation process. After studying the current strategies, she developed a kind of keyword-driven testing and a framework to support it. In 2009 the framework was extended to support also manual testing. Seretta referred about it at EuroSTAR and got the attention of Dorothy Graham that subsequently invited her to contribute with a chapter in her new book (Experiences of Test Automation). On reading her bonus book Seretta noticed recurring patterns in the solution of automation problems. After gaining Dorothy's support, she is currently intent on cataloguing Test Automation Patterns.

Dorothy Graham

Dorothy Graham has been in software testing for 40 years, and is co-author of 4 books: Software Inspection, Software Test Automation, Foundations of Software Testing and Experiences of Test Automation. She has been on the boards of conferences and publications in software testing, was a founder member of the ISEB Software Testing Board and was a member of the working party that developed the ISTQB Foundation Syllabus. She was awarded the European Excellence Award in Software Testing in 1999 and the first ISTQB Excellence Award in 2012. She is currently working on the Test Automation Patterns wiki with Seretta Gamba.

Paul Gerrard

The testers and coding debate: Can we move on now?

Should Testers Learn How to Write Code?

The debate on whether testers should learn how to write code has ebbed and flowed. There have been many blogs on the subject both recent and not so recent. I have selected the ten most prominent examples and listed them below. I could have chosen twenty or more. I encourage you to read references [1, 2, 3, 4, 5, 6, 7, 8, 9, 10].

At the BCS SIGIST in London on the 5th December 2013, a panel discussion was staged on the topic "Should software testers be able to code?" The panellists were: Stuart Reid, Alan Richardson, Dot Graham and myself. Dot recorded the session and has very kindly transcribed the text of the debate. I have edited my contributions to make more sense than I appear to have made 'live'. (I don't know if the other contributors will refine their content and a useful record will emerge). Alan Richardson has captured some pre- and post-session thoughts here – "SIGIST 2013 Panel – Should testers be able to code? [11]. I have used some sections of the comments I made at the session in this article.

It's easy to find thoughtful comments on the subject of testers and coding skills. But why are smart people still writing about the subject? Hasn't this issue been resolved yet? There's a certain amount of hand-wringing and polarisation in the discussion. For example, one argument goes, if you learn how to code, then either:

a) You are not, by definition, a tester anymore; you are a programmer and
b) By learning how to code, you may go native, lose your independence and become a less effective tester.

Another perfectly reasonable view is that you can be a very effective tester without knowing how to code if your perspective is

black-box or functional testing only.

I'd like to explore in this article how I think the situation is obviously not black-and-white. It's what you do, not what you know, that frames your role but also that adding one skill to your skills-set does not reduce the value of another. I'd like to move away from the 'should I, shouldn't I' debate and explore how you might acquire capabilities that are more useful for you personally or your team – if your team need those capabilities.

The demand for coding skills is driven by the demand for capabilities in your project. In a separate article I'll be proposing a 'road-map' for tester capabilities that require varying programming kills.

My Contribution to the 'Debate'

Before we go any further, let me make a few position statements derived from the Q&A of the SIGIST debate. By the way, when the SiGIST audience were asked, it appeared that more than half confirmed that they had programming skills/experience.

Software testers should know about software, but don't usually need to be an expert

Business acceptance testers need to know something of the business that the system under test will support. A system tester needs to know something about systems, and systems thinking. Software testers ought to know something about software, shouldn't they? Should a tester know how to write code? If they are looking at code figuring ways to test it, then probably. And if they need to write code of their own or they are in day to day contact with developers helping them to test their code then technical skills are required. But what a tester needs to know depends on the conversations they need to have with developers.

Code comprehension (reading, understanding code) might be all that is required to take part in a technical discussion. Some programming skills, but not necessarily at a 'professional programmer level', are required to create unit tests, services or GUI test automation, test data generation, output scanning, searching and filtering and so on. The level of skill required varies with the task in hand.

New skills only add, they can't subtract

There is some resistance to learning a programming language from some testers. But having skills can't do you any harm. Having them is better than not having them; new skills only add, they don't subtract.

Should testers be compelled to learn coding skills?

Most of us live in free countries, so if your employer insists and you refuse, then you can find a job elsewhere. But is it reasonable to compel people to learn new skills? It seems to me that if your employer decides to adopt new working practices, you can resist the change on the basis of principle or conscience or whatever, but if your company wishes to embed code-savvy testers in the development teams it really is their call. You can either be part of that change or not. If you have the skills,

you become more useful in your team and more flexible too of course.

How easy is it to learn to code? When is the best time to learn?

Having any useful skill earlier is better than later of course, but there's no reason why a dyed-in-the-wool non-techy can't learn how to code. I suppose it's harder to learn anything new the older you are, but if you have an open mind, like problem-solving, precise thinking, are a bit of a pedant and have patience – it's just a matter of motivation.

However, there are people who simply do not like programming or find it too hard or uncomfortable to think the way a programmer needs to think. Some just don't have the patience to work this way. It doesn't suit everyone. The goal is not usually to become a full time programmer, so maybe you have to persist. But ultimately, it's your call whether you take this path.

How competent at coding should testers be?

My thesis is that all testers could benefit from some programming knowledge, but you don't need to be as 'good' a programmer as a professional developer in order to add value. It depends of course, but if you have to deal with developers and their code, it must be helpful to be able to read and understand their code. Code comprehension is a less ambitious goal than programming. The level of skill varies with the task in hand. There is a range of technical capabilities that testers are being asked for these days, but these do not usually require you to be professional programmer.

Does knowing how to code make you a better tester?

I would like to turn that around and say, is it a bad thing to know how to write code if you're a tester? I can't see a downside. Now you could argue: if you learn to write code, then you're infected with the same disease that the programmers have – they are blind to their own mistakes. But testers are blind to their own mistakes too. This is a human failing not unique to developers of course.

Let's take a different perspective: If you are exploring some feature, then having some level of code knowledge could help you to think more deeply about the possible modes (the risks) of failure in software and there's value in that. You might make the same assumptions, and be blind to some assumptions that the developer made, but you are also more likely to build better mental models and create more insightful tests.

Are we not losing the tester as a kind of proxy of the user?

If you push a tester to be more like a programmer, won't they then think like a programmer, making the same assumptions, and stop thinking of or like the end user?

Dot Graham suggested at the SiGIST event, *"The reason to separate them (testers) was to get an independent view, to find the things that other people missed. One of the presentations at EuroSTAR (2013) was a guy from an agile team who found that all of the testers had 'gone native' and were no longer finding bugs important to users. They had to find a way to get independence back."*

On the other hand, by separating the testers, the team lose much of the rapid feedback which is probably more important than 'independence'. Independence is important, but you don't need to be in a separate team (with a bureaucratic process) to have an independent mind – which is what really matters. The independence, wherever the tester is based, is their independent mind whether it's at the end or working with the developer before they write the code.

There is a Homer Simpson quote [12]: *"How is education supposed to make me feel smarter? Besides, every time I learn something new, it pushes some old stuff out of my brain. Remember when I took that home winemaking course, and I forgot how to drive?"*

I don't think that if you learn how to code, you lose your perspective as a subject matter expert or experience as a real user, although I suppose there is a risk of that if you are a cartoon character. There is a risk of going native if, for example, you are a tester embedded with developers. By the same token, there is a risk that by being separated from developers you don't treat them as members of the same team, you think of them as incompetent, as the enemy. A professional attitude and awareness of biases are the best defences here.

Why did we ever separate testers from developers? Suppose that today, your testers were embedded and you had to make a case that the testers should be extracted into a separated team. I'm not sure the case for 'independence' is so easily made because siloed teams are being discredited and discouraged in most organisations nowadays.

What is this shift-left thing?

There seem to be a growing number of companies who are reducing their dependency on scripted testing. The dependency on exploratory testers and of testers 'shifting left' is increasing.

Right now, a lot of companies are pushing forward with shift-left, Behaviour-Driven Development, Acceptance Test-Driven Development or Test-Driven Development. In all cases, someone needs to articulate the examples – the checks – that drive these processes. Who will write them, if not the tester? With ATDD, BDD approaches, communication is supported with stories, and these stories are used to generate automated checks using tools.

Companies are looking to embed testers into development teams to give the developers a jump start to do a better job (of development and testing). An emerging pattern is that companies are saying, "The way we'll go Agile is to adopt TDD or BDD, and get our developers to do better testing. Obviously, the developers need some testing support, so we'll need to embed some of our system testers in those teams. These testers need to get more technical."

One goal is to reduce the number of functional system testers. There is a move to do this – not driven by testers – but by development managers and accountants. Testers who can't do anything but script tests, follow scripts and log incidents – the plain old functional testers – are being offshored, outsourced, or squeezed out completely and the shift-left approach supports that goal.

How many testers are doing BDD, ATDD or TDD?

About a third of the SIGIST audience (of around 80) raised their hands when asked this. That seems to be the pattern at the moment. Some companies practicing these approaches have never had dedicated independent testers so the proportion of companies adopting these practices may be higher.

Shouldn't developers test their own code? Glen Myers' book [12] makes the statement, "A programmer should avoid attempting to test his or her own program". We may have depended on that 'principle' too strongly, and built an industry on it, it seems. There are far too many testers who do bureaucratic paperwork shuffling – writing stuff down, creating scripts that are inaccurate and out of date, processing incidents that add little value etc. The industry is somewhat bloated and budget-holders see them as an easy target for savings. Shift-left is a reassessment and realignment of responsibility for testing.

Developers can and must test their own code. But that is not ALL the testing that is done, of course.

Do testers need to re-skill?

Having technical skills means that you can become a more sophisticated tester. We have an opportunity, on the technical side, working more closely – pairing even – with developers. (Although we should also look further upstream for opportunities to work more closely with business analysts).

Testers have much to offer to their teams. We know that siloed teams don't work very well and Agile has reminded us that collaboration and rapid feedback drive progress in software teams. But who provides this feedback? Mostly the testers. We have the right skills and they are in demand. So although the door might be closing on 'plain old functional testers' the window is open and opportunities emerging to do really exciting things. We need to be willing to take a chance.

We're talking about testers learning to code but what about developers learning to test better? Should organizations look at this?

Alan Richardson: *We need to look at reality and listen to people on the ground. Developers can test better, business analysts can test better – the entire process can be improved. We're discussing testers because this is a testing conference. I don't know if other conferences are discussing these things, but developers are certainly getting better at testing, although they argue about different ways of doing it. I would encourage you to read some of the modern development books like "Growing Object-Oriented Software Guided by Tests" [14] or Kent Beck [15]. That's how developers are starting to think about testing, and this has important lessons for us as well.*

There is no question that testers need to understand how test-driven approaches (BDD, TDD in particular) are changing the way developers think about testing. The test strategy for a system and testers in general must take account (and advantage) of these approaches.

Summary

In this article, I have suggested that:

- Tester programming skills are helpful in some situations and having those skills would make a tester more productive
- It doesn't make sense to mandate these skills unless your organization is moving to a new way of working, e.g. shift-left
- Tester programming skills rarely need to be as comprehensive as a professional programmer's
- A tester-programming training syllabus should map to required capabilities and include code-design and automated checking methods.

We should move on from the 'debate' and start thinking more seriously about appropriate development approaches for testers who need and want more technical capabilities.

AUTHOR

Paul Gerrard

Paul Gerrard is a consultant, teacher, author, webmaster, developer, tester, conference speaker, rowing coach and a publisher. He has conducted consulting assignments in all aspects of software testing and quality assurance, specialising in test assurance. He has presented keynote talks and tutorials at testing conferences across Europe, the USA, Australia, South Africa and occasionally won awards for them.

Educated at the universities of Oxford and Imperial College London, in 2010, Paul won the Eurostar European Testing excellence Award. In 2012, with Susan Windsor, Paul recently co-authored "The Business Story Pocketbook".

He is Principal of Gerrard Consulting Limited and is the host of the UK Test Management Forum and the UK Business Analysis Forum.

Mail: paul@gerrardconsulting.com
Twitter: @paul_gerrard
Web: gerrardconsulting.com

REFERENCES

[1] Do Testers Have to Write Code?, Elizabeth Hendrickson, http://testobsessed.com/2010/10/testers-code/

[2] Cem Kaner, comments on blog above http://testobsessed.com/2010/10/testers-code/comment-page-1/#comment-716

[3] Alister Scott, Do software testers need technical skills?, http://watirmelon.com/2013/02/23/do-software-testers-need-technical-skills/

[4] Markus Gartner, Learn how to program in 21 days or so, http://www.associationforsoftwaretesting.org/2014/01/23/learn-how-to-program-in-21-days-or-so/

[5] Schmuel Gerson, Should/Need Testers know how to Program, http://testing.gershon.info/201003/testers-know-how-to-program/

[6] Alan Page, Tear Down the Wall, http://angryweasel.com/blog/?p=624, Exploring Testing and Programming, http://angryweasel.com/blog/?p=613,

[7] Alessandra Moreira, Should Testers Learn to Code? http://roadlesstested.com/2013/02/11/the-controversy-of-becoming-a-tester-developer/

[8] Rob Lambert, Tester's need to learn to code, http://thesocialtester.co.uk/testers-need-to-learn-to-code/

[9] Rahul Verma, Should the Testers Learn Programming?, http://www.testingperspective.com/?p=46

[10] Michael Bolton, At least three good reasons for testers to learn how to program, http://www.developsense.com/blog/2011/09/at-least-three-good-reasons-for-testers-to-learn-to-program/

[11] Alan Richardson, SIGIST 2013 Panel - Should Testers Be Able to Code, http://blog.eviltester.com/2013/12/sigist-2013-panel-should-testers-be.html

[12] 50 Funniest Homer Simpson Quotes, http://www.2spare.com/item_61333.aspx

[13] Glenford J Myers, The Art of Software Testing

[14] Steve Freeman and Nat Pryce , Growing Object-Oriented Software Guided by Tests, http://www.growing-object-oriented-software.com/

[15] Kent Beck, Test-Driven Development by Example

[16] A Survey of Literature on the Teaching of Introductory Programming, Arnold Pears et al., http://www.seas.upenn.edu/~eas285/Readings/Pears_SurveyTeachingIntroProgramming.pdf

Tom Gilb

Competitive Planning

Chapter 1 - Objectives

'Objectives' are the plans we have for what we want to achieve, independently of which 'strategies' ('means') we might later select, to achieve them.

All critical objectives can be quantified, and must be

An objective is a valued future performance level, usually an improvement over the current state.

The fact that we can use words like 'enhanced, improved, better' to describe our interests, is a clear sign that these objectives are variable in nature, and that they can be represented by numbers.

There are two primary steps to quantification. First a Scale of Measure needs to be defined. Then interesting levels (like Past, Goal, Tolerable) of that quantified scale, need to be specified.

There are three basic categories of performance objectives: work capacity, savings, and qualities. Most management needs little instruction in quantified specification of the first two of these. But usually needs considerable help in dealing with the other side of the unbalanced scorecard, qualities. Qualities describe 'how well' a system (organization, process, project, product, service) performs.

EXAMPLE

Responsiveness:

Scale: Hours needed for defined People to Correctly Respond to defined Situations.
Goal: within 24 Hours:
- When = End Next Year, People = Director Level, Correctly = Legally & Without Complaint, Respond = Take Action resolving Situation.

Fig 1. *Clear quantified objectives*

We have found no exceptions; all objectives can be quantified.

All word-only objectives, like 'world class quality', or 'enhanced responsiveness to market dynamics' will be unclear to the originator, and will have quite different interpretations to all who read or hear them. They are a total waste of time.

Use the simple basic format, in this example (Figure 1).

Policy 1.1: 'Clear Quantified Objectives' Policy

All critical planning objectives will be expressed with defined Scales of Measure and Numeric Levels.

Why ?
- Force ourselves to think deeply and clearly
- No management bullshit
- Taking responsibility
- Clarifying limits to responsibility

1.2 HANDSFUL: It is sufficient to promote up to ten critical objectives, at any given level of responsibility

There are far more than 10 things we would all like to improve. But if we try to identify and work on 100 or more things at the same time, we will likely lose focus on the more-critical things. We believe and practice that any given level of responsibility (project manager, CTO, IT Architect - for example) should consciously limit themselves to a handful of the most critical objectives, initially. When these are accomplished, or at least safely delegated to others, and on their way to being reached: then it is time to turn to the next set of priorities.

The initial way we do this is in a meeting of people who we need to get to agree what is important. We ask them to list the names of the most critical objectives, and to decide what the top 10, maximum, are. This is about 1 hour of meeting time usually, and it is not too difficult to get pretty good agreement.

Naming the objectives is just the first stage of definition. An it is unreasonable to expect serious commitment to these until

they have been so well defined, quantified, that people know exactly what they have agreed to prioritize. That process can take the rest of the day, of hard parallel work. Three people working on 3 of the objectives, and 3 or 4 teams collating their definitions by the end of the day. But that process works well, and we use it in our 'Evo' Startup Planning Week, process. We achieve the top ten quantified, on a single page - if we edit it that way, in the first day of work on a project.

Bill, a banking VP from New York, asked his boss in London, Barney, the main objective for the startup planning week. Barney replied: "I'd be really overjoyed if for the first time in this Bank's history we managed to quantify, and thus clarify, the primary objectives of our large IT projects."

EXAMPLE

Fig. 2. *Very simplified presentation of top 10 quantified objectives for a client project. The necessary 'Scales of Measure' (see 1.1 above) are not included here, but are implied as defined.*

Bill privately decided to spend an extra day, with Kai, making sure the quantified objectives were top notch, for his boss. While we started work on the top 10 strategies in parallel, on Tuesday.

POLICY 1.2 Top-Ten Critical Objectives Policy

The first day of any project or major effort, will decide, for the moment, on the top 10 most critical objectives: and quantify them on a single page, for responsible management approval.

Why?

- Because all other effort (strategies, estimates) is logically impossible without this clear basis to work from.

What if we need to change the top ten?

- Do that, when and as you need to. But do not use inevitable changes and insights as an excuse for initially fuzzy objectives.

1.3 SUPPORT SUPERIORS: Your level of objectives must clearly support the level above you

Ralph Keeney proposed an excellent practical idea to sort out your responsibility, from your bosses, and your support team's responsibilities. Your objectives ('strategic') must clearly support the achievement of the next level of objectives above you (your bosses objectives, 'fundamental').

Any objectives that presume to support your strategic objectives, your subordinates, or support teams, are called 'means objectives.

EXAMPLE

Contract Flexibility:

Type: Project level Critical Objective.
Owner: Project Manager
Supports: CTO Objectives, especially Technical Adaptability.
Scale: The Speed which a Contract can be Changed at minimum cost of loss to reflect Circumstances.
Goal: < 1 month

- Contract: All IT Services and IT Products
- Changed: Deleted or modified
- Circumstances: changed economics, or failure to live up to expectations
- Deadline: This Year

Supporting Strategies:

- FlexiCon: www.FlexibleContracts.com

Supporting Objectives:

- Legal Dept: % of Flexible Contracts in Force.

Fig 3. *Strategic objectives*

Policy 1.3: Responsibility Clarification

Written specification, immediately tied to objectives, shall clarify the level of responsibility (for formulation, changes and result delivery), as well as what it supports (explicitly defined) and what supports it (explicitly defined).

- Why?

Nobody should be in doubt about their responsibility and its limitations, People should not confuse ends (priority) with means (far less priority).

1.4 LOYAL SUBORDINATION: All your subordinate's objectives must clearly support your objectives

We typically have many and varied sources of support for reaching our own objectives. Direct subordinates, contractors, consultants etc. Let us call any instance that helps you to reach your own objectives, your 'support team'.

You will agree that clarity of responsibility, about how they support your objectives, is necessary. This has some implications.

- If they do not know exactly what your objectives are, they cannot support you very effectively
- If you change, even some details of your objectives: they should be informed, so they can change their support correspondingly
- If you choose to hide your objectives, or to formulate them unclearly: then you are responsible for your support team's lack of ability to serve your interests.
- If you choose to tell them 'what' to do (the means to your objective), rather than the smarter option 'how well to do it' (in terms of your objectives); you bear responsibility for that choice, so be conscious of it. Normally, let them figure out 'how'!

However, once you have made your objective excruciatingly clear, your support team can and should be held accountable, in various ways:

- They should agree, or clearly disagree, that they will support reaching some of your goals, to some degree
- They should be able to show a credible (numeric, experiential, guaranteed) relationship between their activity and plans, and their hope of helping you reach your strategic objectives.
- They should be able to show measurable numeric progress, at least using leading indicators, that their plans are working in practice
- They should expect credibility and rewards, based, not on what they have done – with good intent – but what they have delivered of your values
- Outside contractors should be prepared to put their money where their mouth is, and base payment on your results, not just their effort.

Policy 1.4: Relevant Support Policy

Any element of support for your objectives, should:

- directly show an estimated relationship to your specific numeric objectives
- be prepared to adjust when your objectives are adjusted

- be evaluated on cost effectiveness and timeliness in helping you reach your objectives

Why?
- So we know what to expect, and who is responsible.

1.5 MEANS TO ENDS: All other plans, by whatever names, must support achievement of your goals, on time

Here is a list of 'all other plans, by whatever names':

- All plans for subcontractors and consultants paid from your budget
- All contracts, and agreements
- All sub-projects and their plans
- All strategic plans
- All meetings, training
- All recruitment, and downsizing in your sphere

We should be able to ask: what is the expected impact on our objectives and our budgets.

If the answer is 'nothing' but we need to do it anyway, then let the reason be known (legal, compliance, image, corporate policy) – and accept a degree of it.

If there is any claim to making a contribution to your objectives, then the hard questions can begin [12 Tough Questions]. The objective of the questions is to make both parties think about what they are expecting, and if it is realistic, or risky.

Policy 1.5: Confront Assumptions Policy

Use clear simple, confrontational, questions to find out which activities are really supporting your objectives seriously.

Why?
- To send a message that you are serious about your objectives
- To motivate your support team to think better and more purposefully
- To provide a better set of facts and assumptions to support a contracting process

1.6 MEASURE REALITY: All objectives with a defined Scale, can and must have sufficient measurement methods, to give knowledge of current levels.

"In physical science the first essential step in the direction of learning any subject is to find principles of numerical reckoning and practicable methods for measuring some quality connected with it.

I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely in your thoughts advanced to the state of Science, whatever the matter may be."

Lord Kelvin, 1893, Lecture to the Institution of Civil Engineers, 3 May 1883

THE 12 TOUGH QUESTIONS

1. NUMBERS
Why isn't the improvement quantified?

2. RISK
What's the risk or uncertainty and why?

3. DOUBT
Are you sure? If not, Why not?

4. SOURCE
Where did you get that from? How can I check it out?

5. IMPACT
How does your idea affect my goals?

6. ALL CRITICAL FACTORS
Did we forget anything critical?

7. EVIDENCE
How do you know it works that way?

8. ENOUGH
Have we got a complete solution?

9. PROFITABILITY FIRST
Are we going to do the profitable things first?

10. COMMITMENT
Who's Responsible?

11. PROOF
How can we be sure the plan is working?

12. NO CURE
Is it no cure, no pay?

Many people mix up or combine the concepts of 'quantification' and 'measurement'. They typically use the lazy excuse, that 'perfect measurement' is too difficult, in order to avoid doing 'quantification'.

Illogical!

There is of course a clear enough distinction between a budget and accounting, between a volt and a voltmeter. But people consistently mix up the concepts, to their disadvantage. So did we, for a while.

Notice Kelvin, in the quote above (which determined the direction of our professional work since about 1965). In a single sentence, Lord Kelvin distinguishes between quantification and measurement twice, and three times in the quotation! This is not by accident.

Quantification alone has great merit, even if you never actually carry out any measurement! A budgeting process makes you think about what can and might happen: even though the actual accounting data might be very different. The budgeting process gives you some constraints you have to respect when real measurement threatens to cause problems. The same distinction holds for forming a scientific or engineering hypothesis, and consequent experimentation to determine it is proven or not. Quantification is, above all, a useful tool in communication between people. Numbers clarify, what words hide and confuse. Having recognized that quantification (in practice, defining a scale of measure, and some interesting points on that scale) alone is useful; we also know that it is usually also useful, sooner or later, to actually observe reality numerically: to measure in practice. This gives essential contact with the real world. If measurement is early and frequent, then we can usually adjust our plans, to be in better contact with reality, and with our objectives and constraints.

Measurement does not have to be 'perfect'. In fact it cannot be literally perfect, as engineers and scientists clearly acknowledge. Kelvin was not fanatic, as you can read. So the question is:

- What exactly is sufficient measurement quality (accuracy, precision, credibility), and what is the lowest cost measurement process, that has satisfactory quality.
- At different stages in the system development process, for different purposes, we can decide to have quite different-measurement processes.
- The choice of measurement process, since it depends on many scalar dimensions, is really an 'engineering design' decision.

Here is a simple example showing the distinction, and the choice of more than one measuring tool, for a single scale (Figure 4.).

Policy 1.6: Plan Measurement Formally, and integrated in planning of objectives

Formal written plans, to measure in practice, will be integrated with the specification of objectives.

Why?

- It makes us consider when we want to measure, and consider different levels of measurement capability, and their

EXAMPLE

Team Cooperation Capability:

Type: CTO Level Organizational Improvement Objective
Ambition: much better and consistent cooperation between team members and between teams in technical projects.
Scale: average % of Project Hours spent with Cooperative Content between Team Components.

- Meter [Early Stages of a Project] samples of logged hours, by Project Manager, monthly, 1 hour of work.
- Meter [Analysis of Completed Projects] Database analysis using student trainees, presenting reports and conclusions.

Goal [within 2 years] < 20%-40%

- Project Hours: as logged in project logs, and charged against a project.
- Cooperative Content: writing or oral activity directed to others, with purpose of sharing and/or getting feedback.
- Team Components: Any people within a Team communicating with each other. Any part of a team communicating outside the project team, with the purpose of learning or sharing.

Fig. 4. *Using scales*

costs.
- It will help avoid excessive measurement.

How?
- The 'Meter' parameter can be used for specification of different types of measuring processes.

1.7 CONCEPTUAL COMPLEXITY: Some objectives are complex, have multiple dimensions, and thus multiple scales describing them.

"Love is a many-splendored thing", the old song says. But height and weight have but one dimension [10, Quality Quantification].

One problem you will have encountered in trying to clarify, or to quantify, objectives, is that there might be no one satisfactory dimension of measurement. There are in fact several you can think of. Which one is the right one? It is tempting to ask. All of them andmore might be the right answer! An old electrical handbook recommended dividing up concepts 'until quantification became obvious'.

Rene Descartes (1596-1650) recommended the same approach (Discourses on the Methods):

To accept nothing as true which is not clearly recognise to be so: that is to say, carefully to avoid precipitation and prejudice in judgments, and to accept in them nothing more than what was presented to

my mind so clearly and distinctly that I could have not have no occasion to doubt it. To divide up each of the difficulties which I examined into as many parts as possible, and as seemed requisite in order that it might be resolved in the best manner possible.

To carry on my reflections in due order, commencing with objects that were the most simple and easy to understand, in order to rise little bylittle, or by degrees, to knowledge of the most complex, assuming an order, even if a fictitious one, among those which do not follow a natural sequence relatively to one another. In all cases to make enumerations so completely and reviews so general that I should be certain of having omitted nothing.

Once the CEO at IBM decided that Usability was the wave of the future, for the new Personal Computers, and Tom was asked to help out by IBM.

Tom suggested quantification of Usability, but it took months before we realized that this was many dimensions, not one. The many dimensional model was adopted by IBM.

A client of ours, asked us to analyze a large failed project (8 years, $160 million, 90 project team members). The CEO had initiated the project to radically improve the 'robustness' of a major product. It was failing too often and too long for major customers.

Their original requirement, which went without being taken seriously, for 8 years, was:

"Rock Solid Robustness" (official specification headline)

There was some further specification about not breaking down too often (2 weeks), and being fixed quickly (10 minutes). Combined with a long list of strategies for achieving this.

Tom's suggestion looked more like this:

Rock Solid Robustness:
Type: Complex Product Quality Requirement.
Includes:
{Software Downtime,
Restore Speed,
Testability,
Fault Prevention Capability,
Fault Isolation Capability,
Fault Analysis Capability,
Hardware Debugging Capability}.

The first 3 were then quantified in less than an hour. They should have all been quantified, and used to drive the project, 8 years earlier. Gradual improvements should have been delivered to the quantified goals in the first months of the project.

Policy 1.7: DECOMPOSE COMPLEX OBJECTIVES

Critical top level objectives shall be decomposed into their elementary quantified components, when this will give better management of the top level objective.

Why ?

- Because this gives more realistic understanding and consequent treatment of essential aspects of the problem.
- It forces people to think more deeply
- It eases the path to quantified manageable objectives

EXAMPLE

Market Adaptability:

Type: Marketing Director Critical Objective.
Stakeholders: Marketing Director, CTO, Product Director, Sales Director.
Owner: Chief Marketing Planner
Expert: Supply Chain Manager
Version: 17 July 20xx, 12:31
Quality Control: last approved 10 June 20xx
Scale: .....
Goal: .....

Fig. 5. *Specificaiton of objectives*

1.8 REFLECT REALITY RAPIDLY: Changing specification of objectives, is a natural and necessary response to insights, feedback, competition, and politics

Just because an objective is written, or it is quantified, does not mean it is 'chiseled in stone'. In fact one reason for writing things down, is to clearly see any changes later. A reason for quantification is to more clearly realize that a numeric change has been made, however small.

Our policy must be that changes will be clearly understood. Even the smallest changes can have large consequences. It is therefore important to be able to sense changes, and take appropriate action quickly.

In one published case study (AT&T, 5ESS system, Communications of ACM) the primary factor was a change in switching system availability from 99.90% to 99.98%. Only 00.08% change in one factor. But the cost was 8 years time, and between 2 to 3,000 people were involved. So, imagine the consequences if you are not numeric ("highest availability") or do not have the 4th digit?

We believe in a number of tactics when planning, to make discrete change clearer. We believe it should ne specified at the detailed level (the objective) not the 'total plan' (where the fact of changes can easily get lost.

We would specify, using some of the ideas on Figure 5.

The Stakeholder list makes us aware of the main players concerned with any changes. They can be informed, and they can review and approve. The Owner is the specification owner for this objective. Nobody else can make an official change, and the Owner is responsible for doing it responsibly; for example by informing stakeholders. The owner might also consult with the domain Expert, before publishing a change.

Version control, for example using a date, time stamp or version number helps sensitize readers to changes in the specification. And a QC date and status reminds us that QC is or is not done.
Not all of this detail has to surface every time you quote or use an objective. But it belongs in the planning database, if your plans are to be taken seriously; and if your need to reflect change quickly could otherwise cause confusion.

One underlying principle implied, and behind all this, that we strongly recommend, is that there be allowed only a single valid version, call it a Master Specification, of any given objective or strategy; to which all related planning must refer. Anything less will descend quickly into chaos and anarchy.

Policy 1.8. Change plans quickly, but responsibly

Serious planning objectives will include information allowing us to change details rapidly, but safely; so that all affected parties are made aware of the changes.

Why?

- Anything less that this and you will continue with obsolete plans, and out of sync decision making; not highly competitive.

1.9 RICH REALITY: A single objective can be specified in any useful number of dimensions of time, space, and events.

It can be dangerous to have a single number to represent your objective.

If you do, then it is, unfortunately, logically necessary, to have the biggest number covering all your needs, forever under all circumstances.
That is a bad idea!

We practice differentiation (like 'market segmentation') of targets (what we are aiming to achieve), and constraints (worst acceptable levels). The simple reason for this differentiation is that we can plan more competitively by clearly separating high-value short-term situations from the 'other 95%', and delivering value quickly.
For example, instead of just specifying

Goal: 20%

We would set different goals for specific segments of time, environment (who, where), and events. Technically we call these 'qualifiers' of the goal level.

Something like:

Goal [Deadline = 1st Release, Market = China, Consumer = Golfer, Assumption = Tax Free Import] 20%

Which can also be written:

Goal: 20%

Deadline = 1st Release
Market = China
Consumer = Golfer
Assumption = Tax Free Import

The statement

[Deadline = 1st Release, Market = China, Consumer = Golfer, Assumption = Tax Free Import]

is a 'qualifier'.

It contains three types of qualifier. *When, What,* and *If'* .

There can be any useful number of these. In particular the 'What' dimension often has 3 to 6 more-detailed dimensions in practice. The 'If dimension' is not always used. But it is almost illogical NOT to use the 'When' dimension: since that would allow fulfilling the objective in infinite time. Time constraints have a powerful influence on our chosen means for satisfying our objectives.

Of course you can, at any time in the planning process, have as many different Goal statements, with as many different qualifiers, as you need. You can plan for any set of long-range, medium-range and short-range levels for the objective as you need, when you feel the need.

The planning detail can emerge in parallel with the value-delivery process of your project. Detailed statements can emerge, as you get feedback from real-life delivery; learning about new markets and stakeholders, that are worth catering for.

We usually, predetermine some of the parameters, but not necessarily all of them, in the 'Scale' definition. So we could for example have used the Scale specification, with 'Scale Parameters': like:

> **Scale:** average annual % of our Corporate Unit Sales for a Market, and a Consumer.

This suggests that we can define any useful combination of Market and Consumer in the target (Goal) and constraint (Tolerable) specifications; as well as in Benchmark (Past).

Policy 1.9. The Smart Differentiation Policy

Specify objectives by detailing clear ideas of when, who, what and 'If', so as to maximize our short-term and longer range competitiveness.

Why?

- To avoid delays to urgent selected stakeholders
- To slice up doable short term action
- To force ourselves to think more clearly, and in more detail about our important stakeholders, and what they really need, and how fast it is worth getting the value delivered to them.

1.10 BUTTERFLY: The slightest numeric or other change in an objective, can trigger surprisingly large consequences, in necessary strategies and their costs.

So, be careful what you ask for you might not need it or be able to afford it.

If the state of the art for uptimes of a (software) system is 99.998 %, what will it cost you to demand the best, a competitive edge, say 99.999% ? Just 00.001% better ?

First, nobody knows! There is only one way to find out. Do it.

As all real engineers know, 100% is not possible in finite , for a known cost. They never seriously plan for perfection. Precisely in highly competitive situations, you are pushing the border, the record. And nobody knows, until it is done.

EXAMPLE

Customer Service Availability:

Scale: % of 24/7 a customer gets a qualified answer without waiting or failing.
Past [Last Year, Our Main Service System] 95% <- Service report
Record [Last Year, Our best competitor] 98% <- Their PR
Record [Worldwide, Last 10 years, Similar Customer Service Systems to Ours] 99.98% <- Industry Surveys
Trend [by Next Year, Based on Last 5 years, Our Main Service System] 93% ?
Trend [Next Year, Our best competitor] 99% ??

Based on these benchmarks – what is a reasonable plan?

Tolerable [by Next Year, Our Main Service System] 99% ? <- Mkt Dir.
Goal [by Next Year, Our Main Service System] 99.5% <- CTO

Fig 6. *Benchmarks*

As we pointed out above (AT&T case), the answer can be shocking (24,000 work years of effort 99.90% to 99.98% availability, and this can only be satisfactory, for extremely deep pockets. Management cannot simply, seriously demand '24/7'.

So in our planning language, we have ways of giving ourselves warnings, and of understanding why we have chosen particular levels of an objective, as our Goals or Tolerable levels. We call these devices 'Benchmarks' (Fig. 6.).

Considering the example on Figure 6 - do the benchmarks (Past, Record, Trend) explain and drive us to the levels of the objectives we have suggested? How does Goal : 96% look to you now? Assuming you want to be a winning competitor.

Policy 1.10 Get Realistic

Base your plans on realistic information about state of the art, and state of competition. Specify that information so that it is integrated into your objectives, and preferably updated

Why?

- So you can derive realistic and competitive plans
- So you can explain and justify your objectives to buy-in and approval instances.
- So you can prevent unanchored (in reality) managers from demanding more than you really need to do, or afford to do, or is possible to do.

AUTHOR

Tom Gilb

Tom Gilb was born in Pasadena in 1940, emigrated to London 1956, and to Norway 1958, where he joined IBM for 5 years, and where he resides, and works, when not traveling extensively.

He has mainly worked within the software engineering community, but since 1983 with Corporate Top Management problems, and since 1988 with large-scale systems engineering (Aircraft, Telecoms and Electronics).

He is an independent teacher, consultant and writer. He has published nine books, including the early coining of the term “Software Metrics” (1976) which is the recognized foundation ideas for IBM CMM/SEI CMM/CMMI Level 4.

He wrote “Principles of Software Engineering Management” (1988, in 2006 in 20th printing), and “Software Inspection” (1993, about 14th printing). Both titles are really systems engineering books in software disguise. His latest book is ‘Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering Management Using Planguage’, published by Elsevier, Summer 2005.

He is a frequent keynote speaker, invited speaker, panelist, and tutorial speaker at international conferences.

He consults and teaches in partnership with his son Kai Gilb, worldwide. He happily contributes teaching and consulting pro bono to developing countries (India, China, Russia for example), to Defense Organizations (UK, USA, Norway, NATO) and to charities (Norwegian Christian Aid and others).

He enjoys giving time to anyone, especially students, writers, consultants and teachers, who are interested in his ideas - or who have some good ideas of their own. He is a member of INCOSE (www.incose.org).

His methods are widely and officially adopted by many organizations such as IBM, Nokia, Ericsson, HP, Intel, Citigroup - and many other large and small organizations.

Website: www.gilb.com

Susumu Sasabe

An overview of the SQuBOK® - Software Quality Body of Knowledge - and its benefits in the context of global collaborations for software quality

Abstract

This article presents an overview of the SQuBOK® guide, a guide book on Software Quality Body of Knowledge, and describes its benefits in the context of global collaborations for software quality. The benefits come from a hybrid integration of regional and international software quality knowledge, a distinct feature of the SQuBOK® guide.

Introduction

Today huge information is everywhere in the world and can be accessible beyond borders in real time by using the internet. However it sometimes exceeds one person's information handling capability and unnecessary searching effort is consumed in the information space. Especially it is difficult for a novice software engineer to find, select, and learn essential information from a wide variety of software quality information without assistance by software quality experts. Hence the establishment of the SQuBOK® becomes very important in order to realize easy and fast

access to a structured valuable knowledge successfully practiced in the software and software-based systems industry.

The first edition of the SQuBOK® guide was originally planned, compiled, reviewed, and published in 2007 by a joint project of SQiP (Software Quality Professionals) group of JUSE (Union of Japanese Scientists and Engineers) and Software Division of JSQC (The Japanese Society for Quality Control) [1]. This project was led by Y. Okazaki and the development process of the SQuBOK® guide is reported in the paper [2]. A brief introductory article of the SQuBOK® guide was presented in the German magazine by G. Fessler [3].

Overview of the SQuBOK® guide

The SQuBOK® guide is a collection of software quality knowledge with guidance by the Japanese software quality experts of philosophies and principles relating to their knowledge. It does not include detailed descriptions of the knowledge; instead it provides summarized descriptions of the knowledge and access information to reference materials, such as books, papers, and international de jure and de facto standards for further reading. Description volume is about 1 to 2 pages per one knowledge item. When you open the SQuBOK® guide you may see the description of each knowledge item at a glance without turning pages.

The SQuBOK® guide is a hybrid integration of software quality knowledge recognized and practiced both in Japan and in the world. The hybrid integration has been achieved by using common information framework and template. The common information framework has an affinity for the structure defined in the international standard ISO/IEC 12207 - Software Life Cycle Processes. The common template of the SQuBOK® guide includes following four elements to describe each knowledge item.

Element 1: Outline of the knowledge item
Element 2: Related topics/ knowledge areas of the knowledge item
Element 3: References of the knowledge item
Element 4: Further readings of the knowledge item

The SQuBOK® guide is organized into following three chapters and five appendices.

Chapter 1: Fundamental Concept of Software Quality
Chapter 2: Software Quality Management
Chapter 3: Software Quality Methods

Appendix A: List of Recommended Readings/Papers
Appendix B: List of References/ Further readings
Appendix C: List of Standards
Appendix D: List of Award-Winning Papers
Appendix E: Index

The SQuBOK® is configured with a hierarchical tree diagram which breaks down software quality knowledge into five layers. There are about 300 knowledge items in the tree. Among them, about 40% of the knowledge items are recognized and practiced in Japan and 60% are internationally recognized and practiced knowledge items.

Layer 1: Categories
Layer 2: Sub-categories
Layer 3: Knowledge Areas
Layer 4: Knowledge Sub Areas
Layer 5: Topics

Benefits from using the SQuBOK® guide

For European people, the SQuBOK® guide provides an idea and inspiration to understand not only traditional Japanese quality approaches, such as Kaizen (continuous improvement)[4] and TQC/TQM (Total Quality Control/Management)[5], which contributed to the Japan's rapid economic growth but also to understand a background of recent agile software development approaches, such as SCRUM[6] and Software Kanban[7], which are inspired and formulated through the principles and practices successfully applied in Japanese quality improvement. The foundation of software quality improvement in the Japanese IT organizations and individuals is based on combining software engineering methodologies and TQM.

Readers of the SQuBOK® guide may receive direct messages written by many Japanese software quality experts, some of which were not disclosed outside their organizations in the past. Important messages, which explain tacit knowledge, are now included in the 380 pages SQuBOK® guide in an integrated and structured way.

Readers of the SQuBOK® guide may select the best knowledge from a collection of knowledge to formulate their own software development processes and software quality management systems which suit their own business environment and project needs.

Certification program for Software Quality Engineers in Japan

The SQuBOK® guide provides a foundation of syllabus for software quality engineers education and training program. It also serves as a guide to develop knowledge and skills competency assessment program in the field of software quality. In Japan the JUSE provides a certification scheme called JCSQE (JUSE Certified Software Quality Engineer) program since 2008. Certifications of Foundation and Intermediate levels are in operation and Advanced level is planned. The SQuBOK® guide is not only for software quality assurance professionals but also for all stakeholders relating to software and software-based systems.

AUTHOR

Susumu Sasabe

Susumu Sasabe is an advisor of the JUSE (Union of Japanese Scientists and Engineers, the Deming Prize establisher) since 2008. He joined NEC Corporation in 1972 and worked in research and development of embedded software for telecommunication network systems. He managed several international joint software development projects with companies in North and South America and Asia and conducted a company-wide quality management system and software process improvement inside the NEC Group for over 10 years. His management activities include application of learning through KAIZEN and TQM (Total Quality Management) approach to software engineering and software product innovation.

He has been a frequent speaker at international conferences on software quality including the World Congress for Software Quality (WCSQ). He is also a member of review group of the first version of SQuBOK (Software Quality Body of Knowledge), a software quality knowledge compilation project in Japan.

In 1999 and 2008, he received the Best Paper Presented Award at the 6th European Conference on Software Quality in Vienna, Austria and the Best Quality Technical Paper Award from the Japanese Society for Quality Control (JSQC), respectively.

Conclusion

Before the SQuBOK® guide was released the software quality principles and practices successfully practiced in Japan were very hard to access and understand from outside Japan. The hybrid integration of the SQuBOK® guide provides the readers with an opportunity for sharing both regional and international software quality knowledge and supports to formulate a tailored strategy of software quality improvement which fits to each region in the world. Cross border collaborations enhance merging into an extended SQuBOK® guide to be shared among software quality engineers in the world. The SQuBOK® guide is a useful tool for global teamwork.

REFERENCES

[1] SQuBOK® Project Team, „Guide to the Software Quality Body of Knowledge - SQuBOK® guide", Ohmsha Ltd., ISBN978-4274501623, 2007

[2] Y. Okazaki, T. Okawa, A. Sakakibara, „SQuBOK® (Software Quality Body of Knowledge) Project - Guide to the SQuBOK® Version 1 Born this way", The 4th World Congress for Software Quality, Bethesda, Maryland, U.S.A., 2008

[3] G. Fessler, „Japanese recipe, SQuBOK®: Toolkit for better applications", (in German) iX Magazine No.2, 2014

[4] M. Imai, „Kaizen: The Key to Japan's Competitive Success", McGraw-Hill, ISBN978-0075543329, 1986

[5] K. Ishikawa, „What is Total Quality Control? The Japanese Way", Prentice Hall. ISBN 0139524339, 1985

[6] K. Schwaber and J. Sutherland, „The Scrum Guide ™", 2013 https://www.scrum.org/Portals/0/Documents/Scrum%20Guides/2013/Scrum-Guide.pdf

[7] D. Anderson, „Kanban: Successful Evolutionary Change for Your Technology Business", Blue Hole Press, ISBN 978-0984521401, 2010

Union of Japanese Scientists and Engineers

SQuBOK® Guide
Guide to the Software Quality Body of Knowledge
Concise Version

http://www.juse.or.jp/software/squbok-eng.html

Magazine

Publisher
VWT Polska Michał Kruszewski
Przy Lasku 8 lok. 52, 01-424 Warszawa
Number NIP 5272137158
Number REGON 142455963

Chief editor
Karolina Zmitrowicz
karolina.zmitrowicz@quale.pl

Deputy chief editor
Krzysztof Chytla
krzysztof.chytla@quale.pl

Editors
Bartłomiej Prędki
bartlomiej.predki@quale.pl
Michał Figarski
michal.figarski@quale.pl

WWW
www.qualemagazine.com
www.quale.pl

Facebook
http://www.facebook.com/qualemagazine

Advertisement
info@quale.pl

Cooperation
If you are interested in cooperating with us, please send us a message:
info@quale.pl

The following graphics have been used:

Cover

Ball Landscape River Winter Rays Fractals Fractal
http://pixabay.com/en/ball-landscape-river-winter-rays-67200/