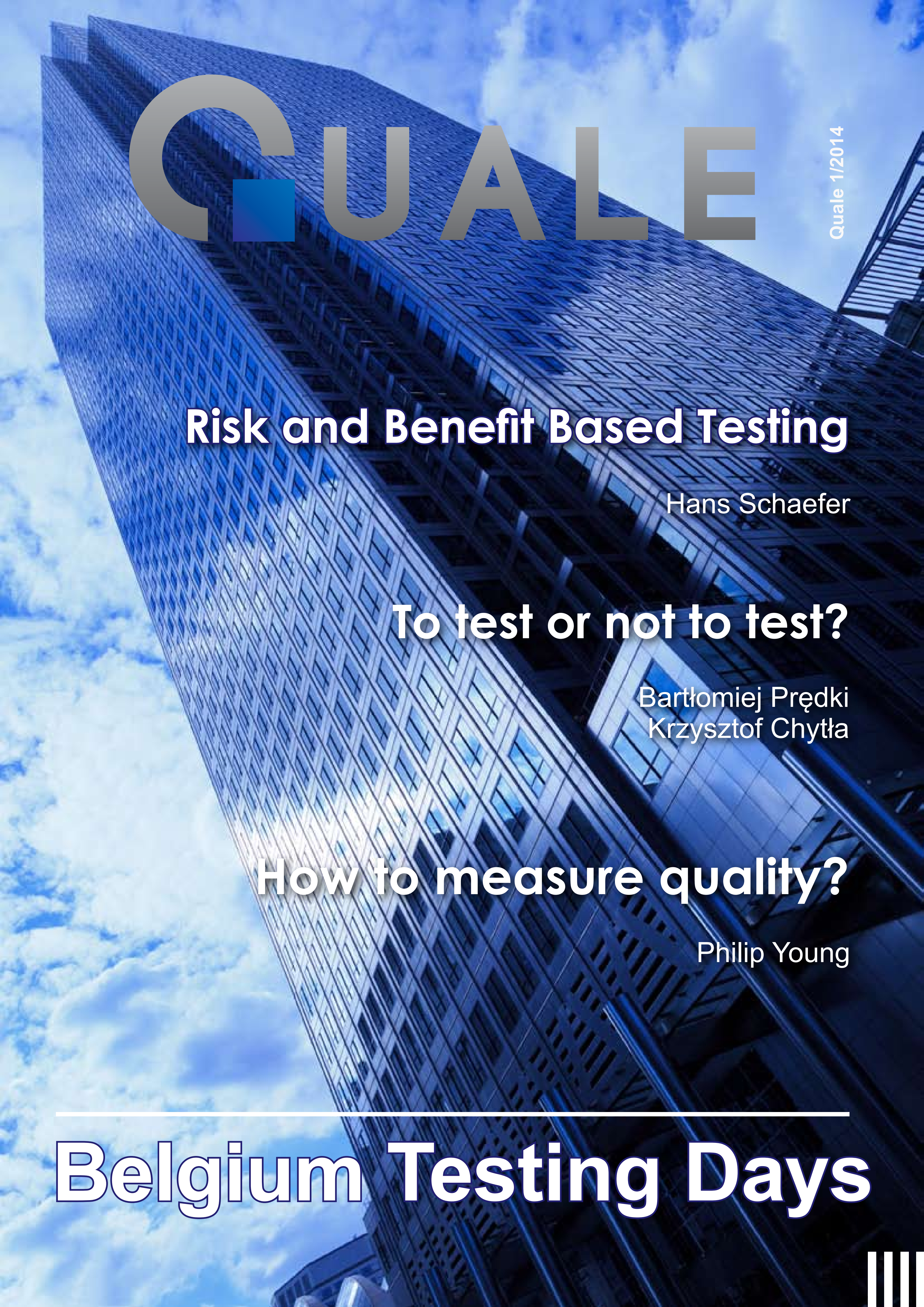
QUALE

Quale 1/2014



Risk and Benefit Based Testing

Hans Schaefer

To test or not to test?

Bartłomiej Prędki
Krzysztof Chytła

How to measure quality?

Philip Young

Belgium Testing Days

Content

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
Piotr Poznański
piotr.poznanski@quale.pl

Website:
www.quale.pl/en (ENG)
www.quale.pl (PL)

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

Bartłomiej Prędki

To test or not to test

That is the question

To test or not to test, that is the question. Let me welcome you, young fellow. As I'm slightly older... I mean a bit more experienced than you, I recently got into existential reflections on our profession. Although it may look as cutting our own throat, I began to wonder, whether we really should do what we do... As you probably know, on almost every software-related training, trainer tries to explain why the testing is necessary, important and valuable, giving various examples of software failures which proved to be fatal. The problem is that the lack of testing in a large part of those failures was not the most important factor. A lot of those failures would have been avoided if project managers had made their job in a proper manner. Let me show you some examples that prove I'm absolutely right.

Reason one. Why trainers do not refer to some spectacular examples where the project was a great success showing an importance of tester's work at the same time? Because it sells poorly, not to mention that testers break the software – so how they can even be considered as the co-creators of success? It's bloody obvious – if something is f***ed up (hate the censorship, by the way), it's better to say "lack of adequate testing" rather than "the management was poor". But if the project ends up with a success, who's praised? Managers (because they managed in an exceptional way) and programmers (you know, those genius coders). No one then remembers testers – the free lunch is for other guys. So basically, that's the way to paint testing black - testers are just needed for someone to blame them for poor results.

Reason two. Let's have a look at wages of programmers and testers. Can you show me the organization, in which testers are paid better than programmers? No? Why? Because such organizations DO NOT EXIST. All this jive about how the testing is „important" is one big hypocrisy. So if testers have to work for peanuts, it's much better to spend that money on a decent training for developers – they will learn how to properly write code and wiii not have to endure the whining of those who can't event reach the middle class... Managers will also be pleased - the budget for testing (which by the way always have to be fought for and explained „why we need this”) can also be spent on more efficient equipment – the faster it compiles, the faster you are able to release. So we have better developers and more efficient hardware – why the hell we need the testing then?

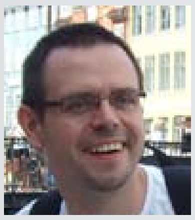
Reason third. Fact: testing everything is impossible. So if we're not able to test everything, then why even start it? It's like getting into the car and immediately knowing that you won't reach the target. Or like starting to watch a movie, knowing from the beginning that you won't see the last scene. So – either we do something right till the end, or we give ourselves (and developers) a break. No self-respecting Project Manager will admit the customer that the system has not been fully tested. So what he does? He blurs the truth behind colorful graphs, showing; of course, only those that leads to the blissful smile of satisfaction on client's face. Hold on, something here just slipped away ... Oh, what - it's unethical? Ethics? In BUSINESS?

Reason four. People like low quality. Many people derive satisfaction from the fact that they have something that breaks down quite often - because then they'll fix it by themselves showing how clever they are. Other ones just need a little bit of sympathy from the world due to the fact that their fate had punished them with such crap.
Reason fifth – the last but not least (there won't be more reasons here as I do not know if anyone has tested the correctness of displaying too longish text in this section of the magazine). As the famous Edward Murphy said, „Anything that can go wrong — will go wrong”.

And no testing will help then.

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 Chytła

To test or not to test

What kind of question is that?

To test or not to test: what kind of questions is that? I need to take a deep breath, calm myself down, hold my horses – yet! - and start the dispute the proper way.

Greetings my jolly good fellow! It's been a while since we last talked or wrote to each other thus I'm sincerely pleased that opportunity has posed itself with such a bold and provocative question. Let me take up the glove. Challenge accepted. Let's test our arguments to drop all doubts once and for all. May the best man win!

Not to test? That's preposterous! Do you remember when a guy "asked for a 13 but they drew a 31" [The Offspring – Pretty fly for a white guy]? A single test would have solved the case.

Primo, let's settle on the concept of who's breaking what. Usually software comes in broken for testing. If not those brave girls and boys committed to bug searching at the cost of their very own eyesight we wouldn't even know that soft in question is – actuall was – broken. Testers are the moral victors of every failed project. It's not about praise and blame (or even blamestorming) but about highest quality available at a reasonable price. A seasoned test specialist will never fall prey to false accusations thanks to the undaunted quality metrics. Nota bene, without testing there would be no relevant quality metrics at all!

Secundo, step by for..., aww just a sec, unfortunately I cannot share confidential HR data with you. That's a pity. However let me give you a small hint: there's no need for company-wide salary revolution but the art of building and developing a dedicated tiger team. All it takes is a profitable business case to back it up.

Remember that „Errāre hūmānum est" [Seneca] which translated into Yoda Testing Language would mean „coder to error prone is" and yes - coders are humans, even those well trained ones. I would go one step further and call a TDD-trained coder a tester! Following that track a tester whose main duty is automation becomes a coder.

Saving on testing? Man please, it is like saving on soap while trying to remove grease. Kindly, take a look at „cost of poor quality" side of the equation. Would you pay for Tetris or Pac Man that does not record high-scores?

Tertio, I agree. (Did I really write that?). That's true. One cannot test everything just like it's not possible to download the whole Internet onto a floppy disk! Let me give you an example. There are millions of computer gamers that just cannot wait, and are willing to queue in order to download 4.7GB of their favorite's sequel through digital distribution channel.

Quattro, there's not accounting for taste. It can be cured these days, you know? I'm more than happy to see some market research on "urging need for poor quality software".
Finally, against Murphy's Law, let me paraphrase one quote: "Bugs! Brace yourselves, Testers are coming!" [R.R. Martin].

Actually there's one more thing. Keep in mind that testing is pride and joy. Or maybe be torment? I guess that's a subject for another discussion.

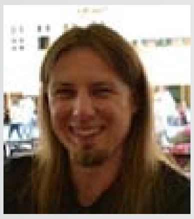
AUTHOR

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.



SQA DAYS

The 15th Anniversary Conference «Software Quality Assurance Days» will hold on April 18-19, 2014 in Moscow, Russia.

We invite you to attend the 15th Anniversary International Conference of Software Quality Assurance – SQA Days.

Once again we have an anniversary. This means that there is opportunity to summarize our results, to reward the best of the best and, of course, to listen to the best speakers. In addition to high quality reports and informative dialogue this year will be a lot of surprises, competitions, installations, and various novelties.

Come and see it yourself. We will be glad to see you!

Hans Schaefer

Risk and Benefit Based Testing

Strategies for prioritizing tests against deadlines

ABSTRACT

Testing is under pressure. Especially test execution has problems, because it is pressed by the collective delays and overruns of the project, even in the short time frame of one sprint. Thus, not everything can and should be tested. Testing should do two things: identify the worst risks and identify important product benefits with low enough risk. Identified risks can be mitigated. Defects may be corrected or published in a known defect list. "Good" product areas are the known benefits. They should show that there is hope for the product. If the risks are too high, more test execution or a reduced feature set may be the result. Just concentrating on testing the risky areas may give an overly pessimistic picture of the product.

Risk-based testing is about prioritizing testing based on product risk. Benefit-based testing is prioritizing areas with high benefits, but maybe less risk.

Not everything is tested to the same depth. Risk contains the possible damage of something not working well enough, as well as the probability that this could happen. Benefit is the necessity to have a feature and the payback from using it. The article shows a method I have been using for many years.

Computing risk

To start with, the possible damage can be found by considering the user's point of view about functions and characteristics of the product. This can be done when writing user stories. It can be classified into categories from „minimal" to „catastrophic". The probability of failing is, in the first run, proportional to the usage frequency. The last step is to estimate the importance of functionality versus nonfunctional attributes. This rough analysis may be used for the first draft test strategy.

However, during the project, more information becomes available. Damage is possible even for project-internal users (for example when components do not get ready in time). Even the usage can be analyzed in more detail. It can be determined how visible some fault may be to external people. Then, for the sake of probability, much more information gets available. It should be more and more known how the project is organized, which people work with which components, where there have been most changes, where the complexity resides etc. Thus, the probability of introducing faults into a product area is not even. The last factor is fault detection through the already planned quality assurance. In areas with thorough checking most defects should be found. In other areas they may survive. All such factors can be considered, in more or less detail, to determine what to test and what not, or what to test more or less.

Computing benefits

Possible benefit can also be found by asking users and customers. It can be classified into categories from „minimal" to „crucial". A benefit is also proportional to the usage frequency. Part of it should be a rating of which components, capabilities and features need to be delivered and working first. As the non-occurrence of a benefit can be classified as the damage part of a risk, concentrating on benefits actually means using risk-based testing without the probability part.

If demonstration of working features is important, high benefit can be combined with low probability for failures from a normal risk calculation.

Integrating the method

The article shows how all this is integrated into a spreadsheet calculation. The method has originally been derived from FMEA (failure modes and effects analysis). The result is normally a classification into three priority classes.

The actual test to be run depends on the available budget, time and how much has been prepared before. In principle, everything should be tested lightly (also called "breadth test"), and risky and high benefit areas should be tested more thoroughly (also called "depth test"). A test method hierarchy is shown.

Furthermore, two special applications are shown:

1. How to prioritize a test when nothing yet is known about the product. Here we take into account that the distribution of faults is uneven. Where there are faults- there are more. Thus, a first

test run is used to prioritize the next one by testing more where faults have been found.
2. What kinds of risks must be considered in the test project, as opposed to the product. It is utmost important to prevent bad quality software from entering testing, as this will increase the time necessary to test. The answer here is to define test entry and exit criteria, and follow up of other quality assurance done.

Disclaimer: The ideas in this paper are not verified for use with safety critical software. Some of the ideas may be useful in that area, but due consideration is necessary. The presented ideas mean that the tester is taking risks, and the risks may or may not materialize in the form of serious failures.

Introduction

The scenario is as follows: You are the test manager. You made a plan and a budget for testing. Your plans were, as far as you know, reasonable and well founded. When the time to execute the tests approaches the product is not ready, some of your testers are not available, or the budget is just cut. You can argue against these cuts and argue for more time or whatever, but that doesn't always help. You have to do what you can with a smaller budget and time frame. Resigning is no issue. You have to test the product as well as possible and you have to make it works reasonably well after release. How to survive?

There are several approaches - using different techniques and attacking different aspects of the testing process. All of them aim at finding as many defects as possible, and as serious defects as possible, before product release. Different chapters of this paper show the idea. At the end some ideas are given that should help to prevent the pressured scenario mentioned before.

In this paper we are talking about the higher levels of testing: integration, system and acceptance test. We assume that developers have done some basic level of testing of every program (unit testing). We also assume the programs and their designs have been reviewed in some way. Still, most of the ideas in this paper are applicable if nothing has been done before you take over as the test manager. It is, however, easier if you know some facts from earlier quality control activities such as design and code reviews and unit testing.

1. The bad game

You are in a bad game with a high probability of loosing: You will lose the game anyway - either by bad testing or by requiring more time to test. After doing bad testing you will be the scapegoat for lack of quality. After reasonable testing you will be the one guilty of late release. A good scenario illustrating the trouble is the Y2K project. Testing may have been done in the last minute, and the deadline was fixed. In most cases, trouble was found during design or testing and system owners were glad that problems were found. In most cases, nothing bad happened after January 1st, 2000. In many cases, managers then decided there had been resources wasted for testing.

But there are options. During this paper I will use Y2K examples to illustrate the major points.

How to get out of the game?

You need some creative solution, namely you have to change the game. You need to inform management about the impossible task you have in such a way that they understand. You need to present alternatives. They need a product going out of the door, but they also need to understand the RISK.

One strategy is to find the right quality level. Not all products need to be free of defects. Not every function needs to work. Sometimes, you have options to do a lot about lowering special product qualities. This means you can cut down testing in less important areas. The typical way to do this is cutting out less needed features. This gives more time to implement the necessary features with high quality.

Another strategy is priority: a test should find the most important defects first. Most important means often "in the most important functions". These functions can be found by analyzing how every one of them supports the mission, and checking which functions are critical and which are not. You can also test more where you expect more defects. Finding the worst areas in the product soon and testing them extensively will help you find more defects. If you find too many serious problems, management will often be motivated to postpone the release or give you more time and resources. The majority of this paper will be about a combination of most important and worst areas' priority.

A third strategy is making testing cheaper in general. One major issue here is automation of test execution. But be cautious: automation can be expensive, especially if you have never done it before or if you do it wrong! However, experienced companies are able to automate test execution with no overhead compared to manual testing. Test automation is crucial in agile projects.

A fourth strategy is getting someone else to pay. Traditionally, this someone else is the customer. You release a lousy product and the customer finds the defects for you. Many companies have applied this. For the customer this game is horrible, as he has no alternative. But it remains to be discussed if this is a good strategy for long term success. So this "someone else" should be the developers, not the testers. You may require the product to fulfill certain entry criteria before you test. Entry criteria can include certain reviews having been done, static analysis, a minimum level of test coverage in unit testing, and a certain level of reliability. The problem is: you need to have high-level support in order to be able to enforce this. Entry criteria tend to be skipped if the project gets under pressure and organizational maturity is low.

The last strategy is prevention, but that only pays off in the next project, when you, as the test manager, are involved from the project start on.

2. Understanding necessary quality levels

Software is embedded in a larger, more complex business world. Quality must be considered in that context [8].

The relentless pursuit of quality can dramatically improve the technical characteristics of a software product. In some applications - medical instruments, railway signaling applications, air-navigation systems, industrial automation, and many defense-related systems - the need to provide a certain level of quality is beyond debate. But is quality really the only or most important framework for strategic decision making in the commercial marketplace?

Quality thinking fails to address many of the fundamental issues that most affect a company's long-term competitive and financial performance. The real issue is which quality attributes will produce the best financial performance.

You have to be sure which qualities and functions are important. Less defects do not always mean more profit! You have to research how quality and financial performance interact. Examples of such approaches include the concept of Return on Quality (ROQ) used in corporations such as AT&T [9]. ROQ evaluates prospective quality improvements against their ability to also improve financial performance. Be also aware of approaches like Value Based Management. Avoid to fanatically pursue quality for its own sake. Define which attributes are crucial. James Whittaker's ACC method [18] may help here.

Thus, more test is not always needed to ensure product success!

Example from the Y2K problem: It may be acceptable that a product fails to work on February 29, 2000. It may also be acceptable that it sorts records wrong if they are blended with 19xx and 20xx dates. But it may be of immense importance that the product could record and process orders after January 1, 2000.

3. Priority in testing: Most important and worst parts of the product

Benefit means the importance of something in the product to the stakeholders. This should be analyzed during working

IMPORTANT

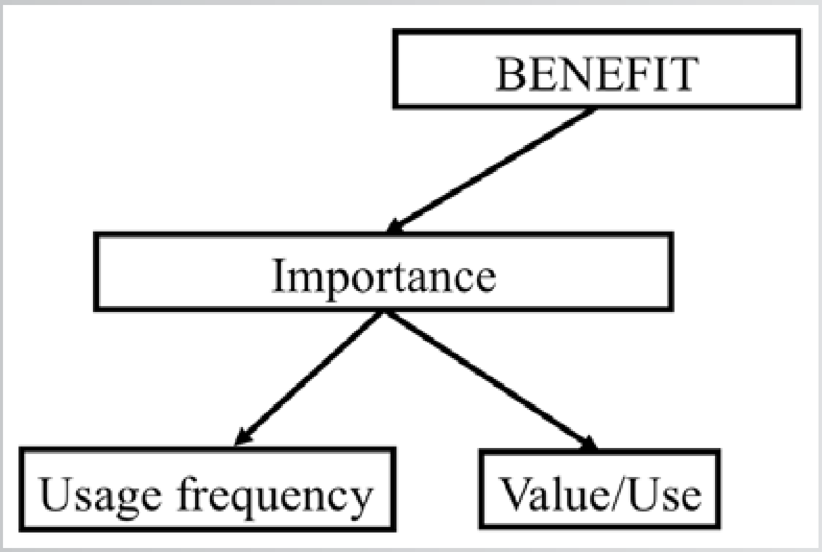


Fig. 1. *Benefit*

with specifications, long before any implementation activities are tried.

Risk is the product of damage and probability for damage to occur. The way to assess risk is outlined in Figure 2 below. Risk analysis assesses damage during use, usage frequency, and determines probability of failure by looking at defect introduction.

Testing is always a sample. You can never test everything and you can always find more to test. Thus, you will always need to make decisions about what to test and what not to test, what to do more or less of. The general goal is to find the worst defects first, the ones that NEED TO BE FIXED BEFORE RELEASE, and to find as many such defects as possible.

This means the defects must be important. The problem with most systematic test methods, like white box testing, or black box methods like equivalence partitioning, boundary value analysis or cause-effect graphing, is that they generate too many test cases, some of which are less important [17]. A way to lessen the test load is finding the most important functional areas and product properties. Finding as many defects as possible can be improved by testing more in bad areas of the product. This means you need to know where to expect more defects.

When dealing with all the factors we look at, the result will always be a list of attributes, components and capabilities with associated importance. In order to make the final analysis as easy as possible, we express all the factors on a scale from 1 to 5. Five points are given for "most important" or "worst", or generally for something having higher risk - which we want to test more - while one point is given to less important areas.

The details of the computation are given later.

3.1. Determining importance or damage: What is important?

You need to know the possible damage resulting from an area to be tested. This

IMPORTANT

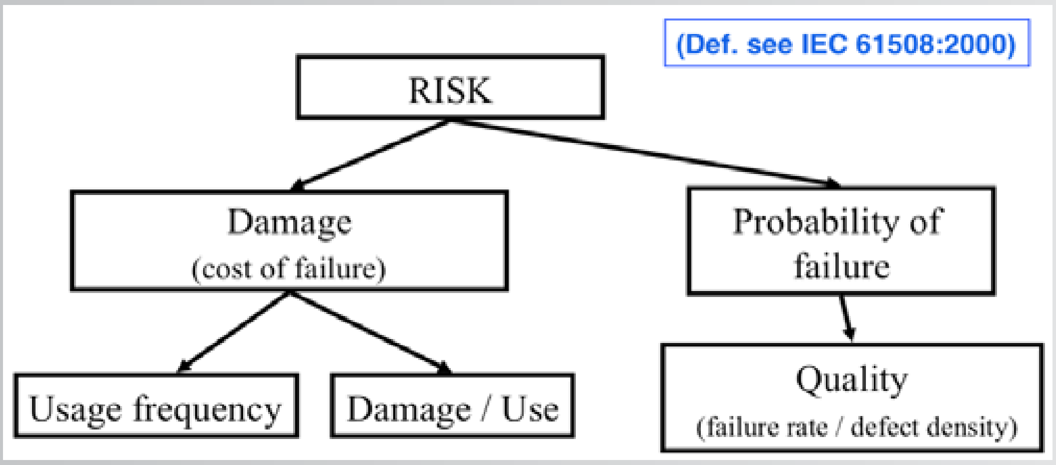


Fig. 2. *Risk*

means analyzing the most important areas of the product. In this section, a way to prioritize this is described. The ideas presented here are not the only valid ones. In every product there may be other factors playing a role, but the factors given here have been valuable in several projects.

Important areas can either be functions or functional groups, or attributes such as performance, capacity, security etc. The result of this analysis is a list of functions and attributes or combination of both that need attention. I am concentrating here on sorting functions into more or less important areas. The approach, however, is flexible and can accommodate other items.

Major factors include:

- Critical areas (importance to the user or cost and consequences of failure)

You have to analyze the use of the software within its overall environment. Find how much the users and other stakeholders value the area. Analyze the ways the software may fail. Find the possible consequences of such failure modes, or at least the worst ones. Take into account redundancy, backup facilities and possible manual check of software output by users, operators or analysts. Software that is directly coupled to a process it controls is more critical than software whose output is manually reviewed before use. If software controls a process, this process itself should be analyzed. The inertia and stability of the process itself may make certain failures less interesting.

> Example: The subscriber information system for a Telecom operator may uncouple subscriber lines - for instance if 31-12-99 is used as «indefinite» value for the subscription end date. This is a critical failure. On the other hand, in a report, the year number may be displayed as blanks if it is in 2000, which is a minor nuisance.

Output that is immediately needed during working hours is more critical than output that could be sent hours or days later. On the other hand, if large volumes of data to be sent by mail are wrong, just the cost of re-mailing may be horrible.

The damage may be classified into the classes mentioned down below, or quantified into money value, whatever seems better. In systems with large variation of damage it is better to use damage as absolute money value, and not classify it into groups.

A possible hierarchy for grouping damage is the following:

A failure would be catastrophic (3)

The problem would cause the computer to stop, maybe even lead to crashes in the environment (stop the whole country or business or product). Such failures may deal with large financial losses or even damage to human life. An example would be the gross uncoupling of all subscribers to the telephone network on a special date.

Failures leading to loosing the license, i.e. authorities closing down the business, are part of this class. Serious legal consequences may also belong here.

The last kind of catastrophic failures is endangering the life of people.

A failure would be damaging (2)

The program may not stop, but data may be lost or corrupted, or functionality may be lost until the program or computer is restarted. An example is equipment that will not work just around midnight on the 31st of December.

A failure would be hindering (1)

The user is forced to workarounds, to more difficult actions to reach the same results.

A failure would be annoying (0)

The problem does not affect functionality, but rather make the product less appealing to the user or customer. However, the customer can live with the problem.

A possible hierarchy for grouping importance is the following:

Crucial (3)

Without this area the product has no value.

Important (2)

Without this area the value of the product is grossly reduced. The customer may just as well choose another product.

Less important (1)

The user may do well without this area, but would value its presence.

Minor (0)

The user may not even notice that this area is not implemented.

CLASSIFICATION

A possible hierarchy for grouping damage is the following:

- A failure would be catastrophic (3)
- A failure would be damaging (2)
- A failure would be hindering (1)
- A failure would be annoying (0)

A possible hierarchy for grouping importance is the following:

- Crucial (3)
- Important (2)
- Less important (1)
- Minor (0)

A possible hierarchy for grouping frequency is the following:

- Unavoidable (3)
- Frequent (2)
- Occasional (1)
- Rare (0)

- Visible areas and risk

The visible areas are areas where many users will experience a failure if something goes wrong. Users do not only include the operators sitting at a terminal, but also final users looking at reports, invoices, or the like, or dependent on the service delivered by the product which includes the software.

A factor to take into account under this heading is also the forgivingness of the users, i.e. their tolerance towards a problem. It relates to the importance of different qualities - see above.

Software intended for untrained or naive users, especially software intended for use by the general public, needs careful attention to the user interface. Robustness will also be a major concern. Software which directly interacts with hardware, industrial processes, networks etc. will be vulnerable to external effects like hardware failure, noisy data, timing problems etc. This kind of software needs thorough validation, verification and retesting in case of environment changes.

An example for a visible area is the functionality in a phone switch, which makes it possible to make a call. Less visible areas are all the value added services like call transfer.

One factor in visibility is possible loss of faith by customers, i.e. longer term damage which would mean longer term loss of business because customers may avoid products from the company.

- Usage frequency

Importance and damage are dependent on how often a function or feature is used.

Some functions may be used every day, other functions only a few times. Some functions may be used by many, some by few users. Give priority to the functions used often and heavily. The number of transactions per day may be an idea helping in finding priorities.

A possibility to leave out some areas is to cut out functionality that is going to be used seldom, i.e. will only be used once per quarter, half-year or year. Such functionality may be tested after release, before its first use. A possible strategy for Y2K testing was to test leap year functionality in January and February 2000, and then again during December 2000 and in 2004.

Sometimes this analysis is not quite obvious. In process control systems, for example, certain functionality may be invisible from the outside. In object oriented systems, there may be a lot of utility libraries used everywhere. It may then be helpful to re-analyze the design of the complete system.

A possible hierarchy is outlined here (from reference [3]):

Unavoidable (3)

An area of the product that most users will come in contact with during an average usage session (e.g. startups, printing, saving).

Frequent (2)

An area of the product that most users will come in contact with eventually, but maybe not during every usage session.

Occasional (1)

An area of the product that an average user may never visit, but that deals with functions a more serious or experienced user will need occasionally.

Rare (0)

An area of the product which most users never will visit, which is visited only if users do very uncommon steps of action. Critical failures, however, are still of interest.

An alternative method to use for picking important requirements is described in [1].

Importance can be classified by using a scale from one to five. However, in some cases this does not sufficiently map the variation of the scale in reality. Then, it is better to use real values, like the cost of damage and the actual usage frequency.

3.2. Failure probability: What is (presumably) worst

The worst areas are the ones having most defects. The task is to predict where most defects are located. This is done by analyzing probable defect generators. In this section, some of the most important defect generators and symptoms for defect prone areas are presented. There exist many more, and you have to always include local factors in addition to the ones mentioned here.

- Complex areas

Complexity is maybe the most important defect generator. Many complexity measures exist, and research into the relation of complexity and defect frequency has been done for more than 30 years. However, no predictive measures have until now been generally validated. Still, most complexity measures may indicate problematic areas. Examples include long modules, many variables in use, complex logic, complex control structure, a large data flow, central placement of functions, a deep inheritance tree, and even subjective complexity as understood by the designers. This means you may do several complexity analyses, based on different aspects of complexity and find different areas of the product that might have problems.

- Changed areas

Change is an important defect generator [13]. One reason is that changes are subjectively understood as easy, and thus not analyzed thoroughly for their impact. Another reason is that changes are done under time pressure and analysis is not completely done. The result is side-effects. Advocates for modern system design methods, like the Cleanroom process, state that debugging during unit test is more detrimental than good to quality, because the changes introduce more defects than they repair.

In general, there should exist a protocol of changes done. This is part of the configuration management system (if something like that exists). You may sort the changes by functional area or otherwise and find the areas which have had exceptionally many changes. These may either have a bad design from before, or have a bad design after the original design has been destroyed by the numerous changes.

Many changes are also a symptom of badly done analysis [5]. Thus, heavily changed areas may not correspond to user expectations.

- Impact of new technology, solutions, methods

Programmers using new tools, methods and technology experience a learning curve. In the beginning, they may generate many more faults than later. Tools include CASE tools, which may be new in the company, or new in the market and more or less unstable. Another issue is the programming language, which may be new to the programmers, or Graphical User Interface libraries. Any new tool or technique may give trouble. A good example is the first project with a new type of user interface. The general functionality may work well, but the user interface subsystem may be full of trouble.

Another factor to consider is the maturity of methods and models. Maturity means the strength of the theoretical basis or the empirical evidence. If software uses established methods, like finite state machines, grammars, relational data models, and the problem to be solved may be expressed suitably by such models, the software can be expected to be quite reliable. On the other hand, if methods or models of a new and unproven kind, or near the state of the art are used, the software may be more unreliable.

Most software cost models include factors accommodating the experience of programmers with the methods, tools and technology. This is as important in test planning as it is in cost estimation.

- Impact of the number of people involved

The idea here is the thousand monkeys syndrome. The more people are involved in a task, the larger is the overhead for communication and the chance that things go wrong. A small group of highly skilled staff is much more productive than a large group of average qualification. In the COCOMO [10] software cost model, this is the largest factor after software size. Much of its impact can be explained from effort going into detecting and fixing defects.

Areas where relatively many and less qualified people have been employed, may be pointed out for better testing.

Care should be taken in that analysis: Some companies [11] employ their best people in more complex areas, and less qualified people in easy areas. Then, defect density may not reflect the number of people or their qualification. Another factor is use of reviews: If reviews are used, the number of involved people increases, but the quality also increases, minimizing risk.

A typical case is the program developed by lots of hired-in consultants without thorough follow-up. They may work in very different ways. During testing, it may be found that everyone has used a different date format, or a different time window.

- Impact of when the work was done

It has been observed in open source projects that components checked-in on Fridays have more bugs than other components (personal communication). Thus, it may be useful to know when some work was done or finished.

- Impact of turnover

If people quit the job, new people have to learn the design constraints before they are able to continue that job. As not everything may be documented, some constraints may be hidden from the new person, and defects result. Overlap between people may also be less than desirable. In general, areas with turnover will experience more defects than areas where the same group of people has done the whole job.

- Impact of time pressure

Time pressure leads to people taking shortcuts. People concentrate on getting the job done, and they often try to skip quality control activities thinking optimistically that everything will go fine. Only in mature organizations this optimism seems to be controlled.

Time pressure may also lead to overtime work. It is well known, however, that people loose concentration after prolonged periods of work. Together with shortcuts in applying reviews and inspections, this may lead to extreme levels of defect density.

Data about time pressure during development can best be found by studying time lists, project meeting minutes, or by interviewing management or programmers.

- Areas which needed optimizing

The COCOMO cost model mentions shortage of machine and network capacity and memory as one of its cost drivers. The problem is that optimization needs extra design effort, or that it may be done by using less robust design methods. Extra design effort may take resources away from defect removal activities, and less robust design methods may generate more defects.

- Areas with many defects before

Defect repair leads to changes which lead to new defects, and defect-prone areas tend to persist. Experience exists that defect-prone areas in a delivered system can be traced back to defect-prone areas in reviews and unit and subsystem testing. Evidence in studies [5] and [7] shows that modules that had faults in the past are likely to have faults in the future. If defect statistics from design and code reviews, and unit and subsystem testing exist, then priorities can be chosen for later test phases.

- Geographical distribution

If people working together on a project are not co-located, communication will be more difficult. This is true even on a local level. Here are some ideas which haven proven to be valuable in assessing if geography may have a detrimental effect on a project:

    - People having their offices in different floors of the same building will not communicate as much as people on the same floor.

- People sitting more than 25 meters apart may not communicate enough.
- A common area in the workspace, such as a common printer or coffee machine improves communication.
- People sitting in different buildings do not communicate as much as people in the same building.
- People sitting in different labs communicate less than people in the same lab.
- As soon as there is more than about two kilometers between office building, people will not meet anymore, but use the phone, email, net meetings or videoconferencing [19].
- People from different countries may have difficulties, both culturally and with the language [19]. If people reside in different time zones, communication will be more difficult. Phone and conference contact depends on overlapping work time. These are problems in distributed or in outsourced software development.

In principle, geographical distribution is not dangerous. The danger arises if people with a large distance have to communicate, for example, if they work with a common part of the system. You have to look for areas where the software structure implies the need for good communication between people, but where these people have geography against them.

- History of prior use

If software has been used before by many users, an active user group can be helpful in testing new versions. Beta testing may be possible. For a completely new system, a user group may need to be defined, and prototyping may be applied. Typically, completely new functional areas are most defect-prone because even the requirements might be unknown or unclear.

- Local factors

Examples include looking at: who did the job, who does not communicate well with someone else, who is new in the project, which department has recently been reorganized, which managers are in conflict with each other, the involvement of prestige and many more factors. Only fantasy sets boundaries. The message is: You have to look out for possible local factors outside the factors having been discussed here.

- One general factor to be considered

This paper is about high level testing. Developers test before this. It is reasonable to take a look at how developers had reviewed and tested the software before and what kind of problems they typically overlook. Analyze the unit test quality. This may lead to further tailoring of the test case selection methods [17].

Looking at these factors will determine the fault density of the areas to be tested. However, using only this will normally overvalue some areas. Typically, larger components will be tested too much. Thus, a correction factor should be applied: functional size of the area to be tested, i.e. the total weight of this area will be “defect proneness / functional volume”. This factor can be found from Function Point Analysis early or from counting code lines if that is available.

What to do if you do not know anything about the project, if all the defect generators can't be applied?

You have to run a test. A first breadth test should find defect-prone areas; the next (depth) test will then concentrate on them. The first test should cover the whole system, but be very shallow. It should only cover typical business scenarios and a few important failure situations, but cover all of the system. You can then find where there was most trouble, and give priority to these areas in the next round of testing. The next round will then do deep and through testing of prioritized areas.

This two-phase approach can always be applied, in addition to the planning and prioritizing done before testing. Chapter 4 explains more of this.

3.3. How to calculate priority of test areas

The general method is to assign weights and to calculate a weighted sum for every area of the system. Test more where the result is the highest!

For every factor chosen, assign a relative weight. You can do this in very elaborate ways, but this will take a lot of time. Most often, three weights are good enough. Values may be: 1, 3, and 10 ("1" for "factor is not very important", "3" for "factor with normal influence", "10" for "factor that has very strong influence").

For every factor chosen, you assign a number of points to every product requirement (every function, functional area, or quality characteristic). The more important the requirement is, or the more alarming a defect generator seems to be for the area, the more points. A scale from 1 to 3 or 5 is normally good enough. Assigning the points is done intuitively.

The number of points for a factor is then multiplied by its weight. This gives a weighted number of points between 1 and 50. These weighted numbers are then summed up for damage (impact) and for

IMPORTANT

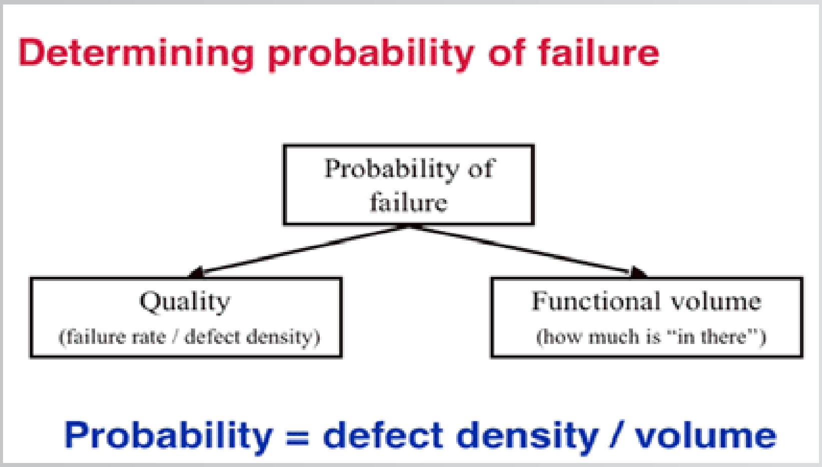

Determining probability of failure
Probability of failure
Quality
(failure rate / defect density)
Functional volume
(how much is "in there")
Probability = defect density / volume

Fig. 3. *Failure Probability*

probability of errors, and finally multiplied (see remark below). Testing can then be planned by assigning most test cases to the areas with the highest number of points.

An example for benefit calculation (functional volume being equal for the different areas) (Tab. 1.).

Table 1 suggests that function «invoicing» is most important to test, «order registration» and performance of order registration are next. The factor which has been chosen as the most important is visibility.

Computation is easy, as it can be programmed using a spreadsheet. A spreadsheet is on http://www.softwaretesting.no/testing/benefitcalc.xls .

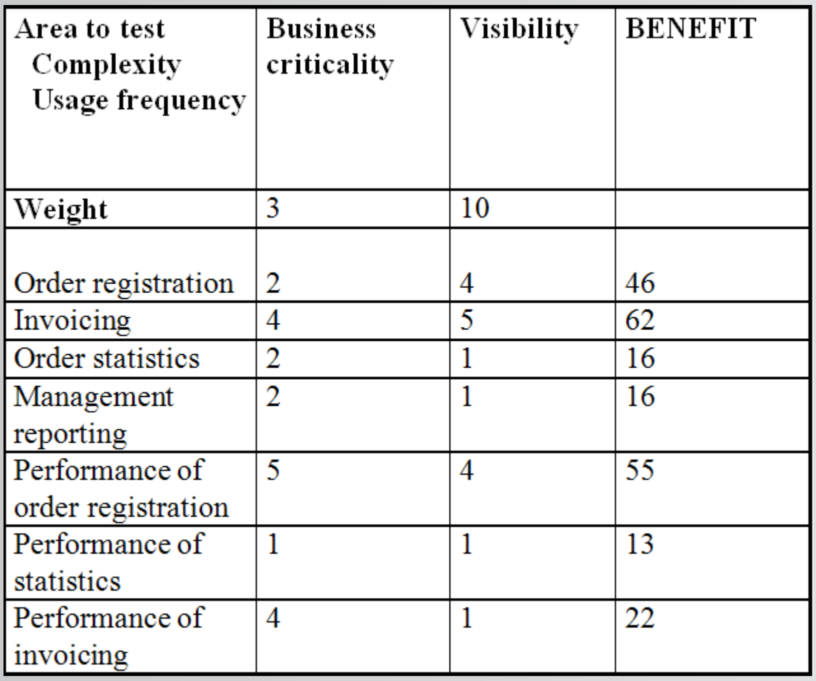
An example for risk calculation (functional volume being equal for the different areas) can be found from (Tab. 2.).

The table above requires you to know something about complexity of areas and their change frequency. This is typically only known later during a project. Table 2 suggests that function «invoicing» is most important to test, then «order registra-

REMARK

As many intuitive mappings from reality for points seem to involve a logarithmic scale, where points follow about a multiplier of 10, the associated risk calculation should ADD the calculated weighted sums for probability and damage. If most factors' points inherently follow a linear scale, the risk calculation should MULTIPLY the probability and damage points. The user of this method should check how they use the method!

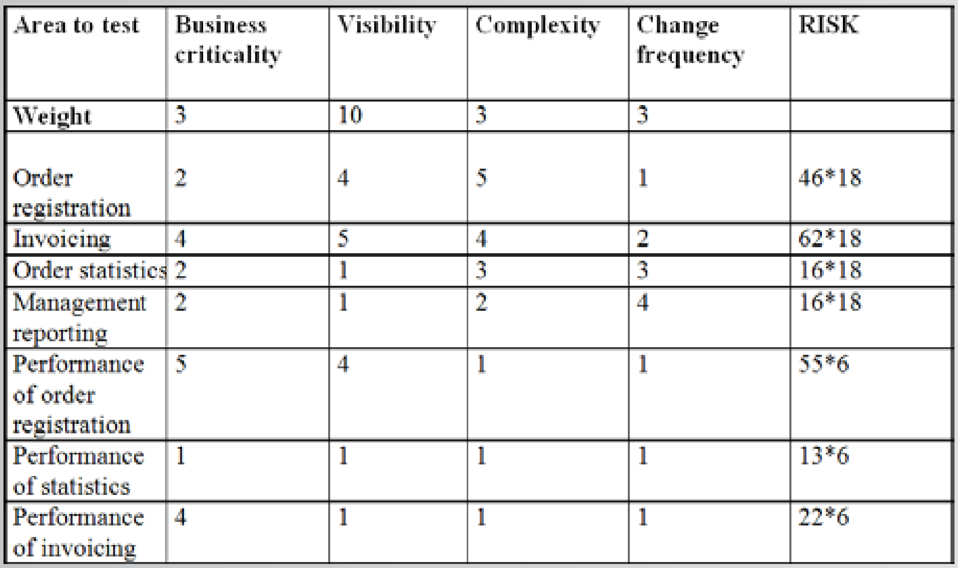
IMPORTANT



Area to test Complexity Usage frequency	Business criticality	Visibility	BENEFIT
Weight	3	10	
Order registration	2	4	46
Invoicing	4	5	62
Order statistics	2	1	16
Management reporting	2	1	16
Performance of order registration	5	4	55
Performance of statistics	1	1	13
Performance of invoicing	4	1	22

Tab. 1. *Benefit calculation*

IMPORTANT



Area to test	Business criticality	Visibility	Complexity	Change frequency	RISK
Weight	3	10	3	3	
Order registration	2	4	5	1	46*18
Invoicing	4	5	4	2	62*18
Order statistics	2	1	3	3	16*18
Management reporting	2	1	2	4	16*18
Performance of order registration	5	4	1	1	55*6
Performance of statistics	1	1	1	1	13*6
Performance of invoicing	4	1	1	1	22*6

Tab.2. *Risk calculation*

tion» and «performance of order registration». The factor which has been chosen as the most important is «visibility».

Computation is easy, as it can be programmed using a spreadsheet. A more detailed case study is published in [4]. A spreadsheet is available on http://www.softwaretesting.no/testing/riskcalc.xls

A word of caution: The assignment of points is intuitive and may be wrong. Thus, the number of points can only be a rough guideline. It should be good enough to distinguish the high-risk areas from the medium and low risk areas. That is its main task. This also means you don't need to be more precise than needed for just this purpose. If more precise test prioritization is necessary, a more quantified approach should be used wherever possible.

4. Making testing more effective

More effective test means to find more and more important defects in the same amount of time.

The strategy to achieve this is to learn from experience and adapt testing.

First, the whole test should be broken into four phases:

- test preparation
- pre-test
- main test
- after-test

Test preparation sets up areas to test, the test cases, test programs, databases and the whole test environment. Especially setting up the test environment can give a lot of trouble and delay. It is generally easy to install the program itself and the correct operating system and database system. Problems often occur with the middleware, i.e. the connection between software running on a client and software running on different servers. Care should be taken to thoroughly specify all aspects of the test environment; and dry runs should be held in order to ensure that the test can be run when it is time to do it. Much can be achieved using virtualization. Nevertheless testing with modern platforms, especially mobile, can make setup of test environments very difficult. Time should be reserved in order to do and check this early on.

In a Y2K project, care was taken to ensure that licenses were in place for machine dates after 1999, and that the licenses allowed resetting of the machine date. Another area to focus was that included software had been Y2K compliant.

The pre-test is run after the software under test is installed in the test lab. This test contains just a few test cases running typical day to day usage scenarios. The goal is to test if the software is ready for testing at all, or totally unreliable or incompletely installed. Another goal may be to find some initial quality data, i.e. find some defect prone areas to focus the further test on. This test MUST be automated in any case.

The main test consists of all the pre-planned test cases. They are run, failures are recorded, defects found and repaired, and new installations of the software made in the test lab. Every new installation may include a new pre-test. The main test takes most of the time during a test execution project.

The after-test starts with every new release of the software. This is the phase where optimization should occur. Part of the after-test is regression testing, done in order to find possible side-effects of defect repair. But the main part is a shift of focus. Exploratory testing is a natural part of this phase.

Type of defects may be analyzed. A possible classification is described in [14]. In principle, every defect is a symptom of a weakness of some designer, and it should be used to actively search for more defects of the same kind.

Example: In a Y2K project, it was found that sometimes programs would display blank instead of zeroes in the year field in year 2000. A scan for the corresponding wrong code through many other programs produced many more instances of the same problem.

Another approach is to concentrate more tests on the more common kinds of defects, as these might be more common in the code. The problem is, however, that such defects might already have been found because the test was designed to find more of this kind of defects. Careful analysis is needed. Generally, apply the abstractions of every defect found as a checklist to more testing or analysis.

Location of defects may also be used to focus testing. If an area of code has especially many failures, that area should be a candidate for even more testing [7, 13]. Moreover during the analysis, care should be taken to ensure that a high level of defects in an area is not caused by especially high test coverage in that area.

5. Making testing cheaper

A viable strategy for cutting budgets and time usage is to do the work in a more productive and efficient way. This normally involves applying technology. In software, not only technology, but also personnel qualifications seem to be ways to improve efficiency and cut costs. This also applies in testing.

5.1. Automation

There exist many test automation tools. Tools catalogues list more tools for every new edition and the existing tools are more and more powerful while not costing more [12]. Automation can probably do most in the area of test running and regression testing. Experience has shown that more test cases can be run for much less money, often less than a third of the resources spent for manual testing. In addition, automated tests often find more defects. This is fine for software quality, but may hit the testers, as the defect repair will delay the project... Still, such tools are not very popular, because they require an investment into training, learning and building an infrastructure at start. Sometimes a lot of money is spent in fighting with the tool. For the productivity improvement, nothing general can be said, as the application of such tools is too dependent on platforms, people and organization. Evaluate your tools wisely; make sure they are fit for the purpose. Anecdotal evidence prevails, and for some projects automation has had a great effect.

An area where test is nearly impossible without automation is stress, volume and performance testing. Here, the question is either to do it automatically or not to do it at all.

Test management can also be improved considerably using tools for tracking test cases, functions, defects and their repairs. Such tools are now more and more often coupled to test running automation tools.

In general, automation is interesting for cutting testing budgets. You should, however, make sure you are organized, and you should keep the cost for startup and tool evaluation outside your project. Tools help only if you have a group of people who already know how to use them effectively and efficiently. To bring in tools in the last moment has a low potential to pay off, and can do more harm than good.

5.2. The people factor - Few and good people against many who know little

The largest obstacle to an adequate testing staff is ignorance on the part of management. Some of them believe that "development requires brilliance, but anybody can be a tester."

Testing requires skill and knowledge. Without application knowledge your testers do not know what to look after. You get shallow test cases which do not find defects. Without knowledge about common errors the testers do not know how to make good test cases (see remark below). Again, they do not find defects. Without experience in applying test methods people will use a lot of unnecessary time to work out all the details in a test plan.

If testing has to be cheap, the best is to get a few highly experienced specialists to collect the test candidates, and have highly skilled testers to improvise the test instead of working it out on paper. Skilled

REMARK

Good test cases, i.e. test cases that have a high probability of finding errors, if there are errors, are also called «destructive test cases».

people will be able to work from a checklist, pick equivalence classes, boundary values, and destructive combinations by improvisation. Non-skilled people will produce a lot of paper before having an even less destructive test. A method for this is called "exploratory testing".

Testers must be at least equally smart, equally good designers and have equal understanding of the functionality of the system as coders. One could let the Function Design Team Leader become the System Test Team Leader as soon as functional design is complete. Pre-sales, Documentation, Training, Product Marketing and/or Customer Support personnel should also be included in the test team. This provides early knowledge transfer (a win-win for both development and the other organization) and more resources than there exist full-time. Test execution requires lots of bodies that don't need to be there all of the time, but need to have a critical and informed eye on the software. You probably also need full-time testers, but not as many as you would use in the peak testing period. Full-time test team members are good for test design and execution, but also for building or implementing testing tools and infrastructure during less busy times.

If an improvised test has to be repeated, there is a problem. But modern test automation tools can be run in a capture mode, and such captured test may later be edited for documentation and rerunning purposes.

The message is: Get highly qualified people for your test team!

6. Cutting testing work

Another way of cutting costs is to get rid of part of the task. Get someone else to pay for it or cut it out completely!

6.1. Who pays for unit testing?

Often, unit testing is done by the programmers and never turns up in any official testing budget. The problem is that unit testing is often not really done. Test coverage tool vendors often report that without their tools, 40 - 50% of the code is never unit tested. Many defects then survive until later test phases. This means later test phases have to test better, and they are overloaded and delayed by finding all the defects which could have been found earlier.

As a test manager, you should require higher standards for unit testing! This is in line with modern "agile" approaches to software development. Unit tests should be automated as well and rerun every time units are changed or integrated.

6.2. What about test entry criteria?

The idea is the same as in contracts with external customers. If the supplier does not meet the contract, the supplier gets no acceptance and no money. Problems occur when there is only one supplier and when there is no tradition in requiring quality. Both conditions are true in software. But entry criteria can be applied if the test group is strong enough. Criteria range from the most trivial to advanced.

Here is a small collection of what makes the life in testing easier:

- The system delivered to integration or system test is complete
- It has been run through static analysis and defects are fixed
- A code review has been done and defects have been corrected
- Unit testing has been done to the accepted standards (near 100% statement coverage, for example)
- Any required documentation is delivered and is of a certain quality
- The units compile and can be installed without trouble
- The units should have passed some functional test cases (smoke test).
- Really bad units are sorted out and have been subjected to special treatment like extra reviews, reprogramming etc.

You will not be allowed to require all these criteria. You may not be allowed to enforce them. However you may turn projects into a better state over time by applying entry criteria. If every unit is reviewed, statically analyzed and unit tested, you will have a lot less problems to fight with later.

6.3. Less documentation

If a test is designed “by the book” it will take a lot of work to document it. Not all this is needed. Tests may be coded in a high level language and may be self documenting. A test log made by a test automation tool may do the service. Qualified people may be able to make a good test from checklists, and even repeat it. Check out exactly which documentation you will need and prepare no more. Most important is a test plan with a description of what is critical to test, and a test summary report describing what has been done and the risk of installation.

6.4. Cutting installation cost - strategies for defect repair

Every defect delays testing and requires an extra cost. You have to rerun the actual test case, try to reproduce the defect, document as much as you can, probably help the designers debugging, and at the end install a new version and retest it. This extra cost is impossible to control for a test manager, as it is completely dependent on system quality. The cost is normally not budgeted for either. Still, this cost will occur. Here is some advice about how to keep it low.

6.5. When to correct a defect, when not?

Every installation of a defect fix means disruption: installing a new version, initializing it, retesting the fix, and retesting the whole. The tasks can be minimized by installing many fixes at once. This means you have to wait for defect fixes. On the other hand, if defect fixes themselves are wrong, this strategy leads to more work in debugging the new version. The fault is not that easy to find. There will be an optimum, dependent on system size, the probability to introduce new defects, and the cost of installation. For a good description of practical test exit criteria, see [2]. Here are some rules for optimizing the defect repair work:

- Rule 1: Repair only important defects!
- Rule 2: Change requests and small defects should be assigned to the next release!

- Rule 3: Correct defects in groups! Normally only after blocking failures are found.
- Rule 4: Use an automated "smoke test" to test any corrections immediately.

7. Strategies for prevention

The starting scenario for this paper is the situation where everything is late and where no professional budgeting has been done. In most organization, there exist no experience data and there exists no serious attempt to really estimate costs for development, testing, and error cost in maintenance. Without experience data there is no way to argue about the costs of reducing a test.

The imperatives are:

- You need a cost accounting scheme
- You need to apply cost estimation based on experience and models
- You need to know how test quality and maintenance trouble interact

Measure:

- Size of project in lines of code, function points, etc.
- Percentage of work used in management, development, reviews, test preparation, test execution, and rework
- Amount of rework during first three or six months after release
- Fault distribution, especially causes of user detected problems.
- Argue for testing resources by weighting possible reductions in rework before and after delivery against added testing cost.

Papers showing how such cost and benefit analysis can be done, using retrospective analysis, have been published in several ESSI projects run by Otto Vinter from Bruel&Kjær [6]. A different way to prevent trouble is incremental delivery. The general idea is to break up the system into many small releases. The first delivery to the customer is the least commercially acceptable system, namely, a system which does exactly what the old one did, only with new technology. From the test of this first version you can learn about costs, error contents, bad areas, etc. - then you have an opportunity to plan better.

8. Summary

Testing in a situation where management cuts both budget and time is a bad game. You have to endure and survive this game and turn it into a success. The general methodology for this situation is not to test everything a little, but to concentrate on high benefit areas and the worst areas. Combine testing things with a high benefit with testing "risky" areas.

SUMMARY

Priority 1: Return the product as fast as possible to the developers, with a list of as serious deficiencies as possible. BUT: Show them that there is hope, by highlighting that most important areas of the product work.

Priority 2: Make sure that whenever you stop testing, you have done the best testing in the time available!

REFERENCES

[1] Joachim Karlsson & Kevin Ryan, "A Cost-Value Approach for Prioritizing Requirements", IEEE Software, Sept. 1997

[2] James Bach, "Good Enough Quality: Beyond the Buzzword", IEEE Computer, Aug. 1997, pp. 96-98

[3] Risk-Based Testing, STLabs Report, vol. 3 no. 5 (info@stlabs.com)

[4] Ståle Amland, "Risk Based Testing of a Large Financial Application", Proceedings of the 14th International Conference and Exposition on TESTING Computer Software, June 16-19, 1997, Washington, D.C., USA.

[5] Tagji M. Khoshgoftaar, Edward B. Allan, Robert Halstead, Gary P. Trio, Ronald M. Flass, "Using Process History to Predict Software Quality," IEEE Computer, April 1998

[6] Several ESSI projects, about improving testing, and improving requirements quality, have been run by Otto Vinter. Contact the author at otv@delta.dk.

[7] Ytzhak Levendel, "Improving Quality with a Manufacturing Process", IEEE Software, March 1991.

[8] "When the pursuit of quality destroys value", by John Favaro, Testing Techniques Newsletter, May-June 1996.

[9] "Quality: How to Make It Pay," Business Week, August 8, 1994

[10] Barry W. Boehm, Software Engineering Economics, Prentice Hall, 1981

[11] Magne Jørgensen, 1994, "Empirical studies of software maintenance", Thesis for the Dr. Scient. degree, Research Report 188, University of Oslo.

[12] Lots of test tool catalogues exist. The easiest accessible key is the Test Tool FAQ list, published regularly on Usenet newsgroup comp.software.testing. More links on the author's web site.

[13] T. M. Khoshgoftaar, E.B. Allan, R. Halstead, Gary P. Trio, R. M. Flass, «Using Process History to Predict Software Quality», IEEE Computer, April 1998

[14] IEEE Standard 1044, A Standard Classification of Software Anomalies, IEEE Computer Society.

[15] James Bach, «A framework for good enough testing», IEEE Computer Magazine, October 1998

[16] James Bach, "Risk Based Testing", STQE Magazine,6/1999, www.stqemagazine.com

[17] Nathan Petschenik, "Practical Priorities in System Testing", in "Software- State of the Art" by DeMarco and Lister (ed), Sept. 1985, pp.18 ff

[18] James Whittaker developed Google's ACC method https://sites.google.com/site/visualisingquality/techniques/acc-matrix, http://www.youtube.com/watch?v=cqwXUTjcabs .

[19] Martins Gills, Marints.gills@riti.lv, "Outsourcing: Distance is relative", Professional Tester Magazine, Sept 2005.

Hans Schaefer

AUTHOR

Specialist in software testing

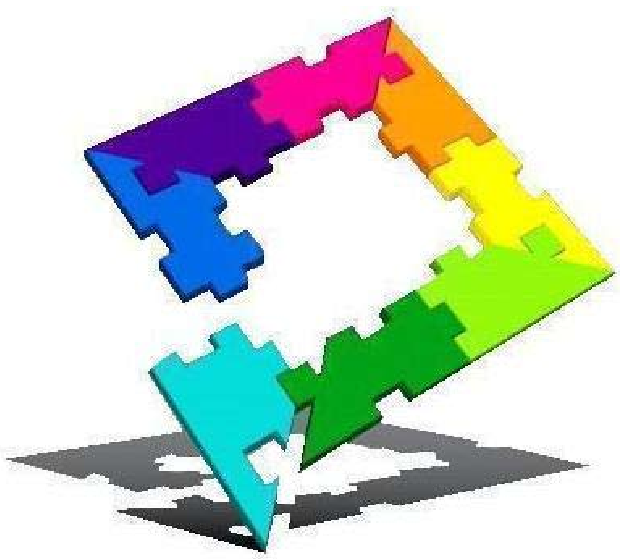
1952 born
1979 M. Eng. from Technical University of Braunschweig, Germany. Computer science, railway signaling
1979-1986 software developer at research facilities in Germany and Norway
1983-1986 Software tester, software quality consultant at SI (now part of SINTEF), Norway
1984-1985 guest lectures about software quality assurance at Oslo university
1985 and later: guest lectures about software testing at several Nordic universities
1987-2014 Consultant in software testing, based in Norway, working all over the world

ISTQB full advanced certified, since 2004 leader of Norwegian Testing Board (ISTQB). Member of ISTQB Foundation Level Working Group, responsible for ISTQB Foundation Syllabi.
Certified Mountain Guide, certified steam locomotive fireman.

I have been running my own company since 1987, specializing in consulting about software testing, reviews and software quality matters. I am teaching seminars about software testing, mostly in Scandinavian countries, Germany and China.

I have worked for most leading Norwegian companies, as well as companies like Bombardier, Ericsson, Nokia, Statoil, Telenor, Visma.

www.softwaretesting.no
hschae@broadpark.no



BELGIUM TESTING DAYS

CONFERENCE 17TH - 20TH MARCH 2014

Philip Young

How to measure quality?

THE PURPOSE OF THIS ARTICLE

Recently at a software meet-up we posed a question of "how do people measure quality" and were surprised to find that nobody really had an answer, or anything really concrete to say on the subject. Indeed in some cases (mainly freelance software development houses), quality was not even on the agenda, working software was shown to imply quality had been achieved).

I found myself in a situation recently where I was able to alter the ways our firm measured quality and chose this moment to try some out some ideas that were new to us. The article describes the problems I faced even just defining what quality actually is and goes on to describe the associated measures that I created around my perception of it. All the measures described below were split by user and by department and presented to upper management on a rolling three monthly basis.

What is Quality? – The great quality debate.

My personal point of view is that Quality is not just "how many bugs exist in live" it extends right the way through the SDLC and throughout every walk of life, but to discuss this further in any detail is impractical for the scope of the article, so here I include only measures for things we can actually control in test and that the firm I work for is interested in.

Just about every person, whether in the testing industry or part of the general public has a different view of what quality is. There immediately exists a problem with this article therefore. How can we measure something if we cannot define what is actually is? It is sometimes easier to answer what quality isn't – we know that good quality is not a buggy piece of software delivered outside of deadline and without informing anyone of its capabilities for example.

Every end product, be it tangible or otherwise has an particular quality level associated with it – consider for example the difference between a Rolls Royce and a Mini – would the build quality be the same in both these cars? Or would it be the right quality for the company that is selling them? Would a customer of these firms say they exceed, match or underperform their expectations in regards to their build quality? What about the quality of individual buttons in the car? Or Ride quality? etc. Quality means many things to many people and this is why we cannot just consider quality to be measured via the number of bugs in live.

We must also consider what fits for the particular organisation we work for to determine what quality actually is. We also need to think about what the perception of quality is to a business that is designed from the ground up to make profit – what level of quality is good enough to a business like that? Quality close to perfection would not be needed in a case like this for example. We are in fact saying then that some bugs are allowable, as long as they are not too severe? But what is severe? If some bugs are allowable, how many? Do we allow the same number of bugs and the severity of them to be the same for different sizes of projects? If not how do we compare these? Can they be compared across projects and smaller changes? We must consider the product as a whole, and include the quality of our processes and estimation as well as bugs into the live environment.

Measuring Quality

I chose to measure quality in three ways:

1) Process

The process referred to below is not a rigid process, we use a flexible tick-list to allow for addition or subtraction of points for process improvement, but the results still remained directly comparable over each month. The results of this were split by user and department each month.

The process measure asks if we are following the process correctly. If we do not follow process, vital communication lines are lost, a department may not be ready for the change in time, the actual program and configuration changes going live may conflict with other changes etc

To measure process, I used a pre-existing document that was already being followed to cover off the above points and identified the most critical points on it. Each of these scored a "1" when the point was adequately covered, so whilst this did introduce an overhead of checking the output from these each month. I knew that under the principles of measurement, once I started measuring this process, it very quickly became followed, so I was able to use this often citied disadvantage of measurement for departmental gain. The measurement of our process was always intended to be a quick win, almost immediately people followed the process more accurately than they had been and if anything it created a positive vibe during the monthly meeting with the worker when I could observe evidence that due process had been followed and present this to upper management as a positive result. The measure is intended to be simple; it just needed to achieve the desired effect described above. Figure 1 shows example output from this measure.

2) Estimation

Estimation at the time was something we were not doing formally, we had an additional challenge as many of our changes are smaller in size (say 1-2 weeks max) so I chose to measure the estimates in hours and recorded them with Microsoft Project.

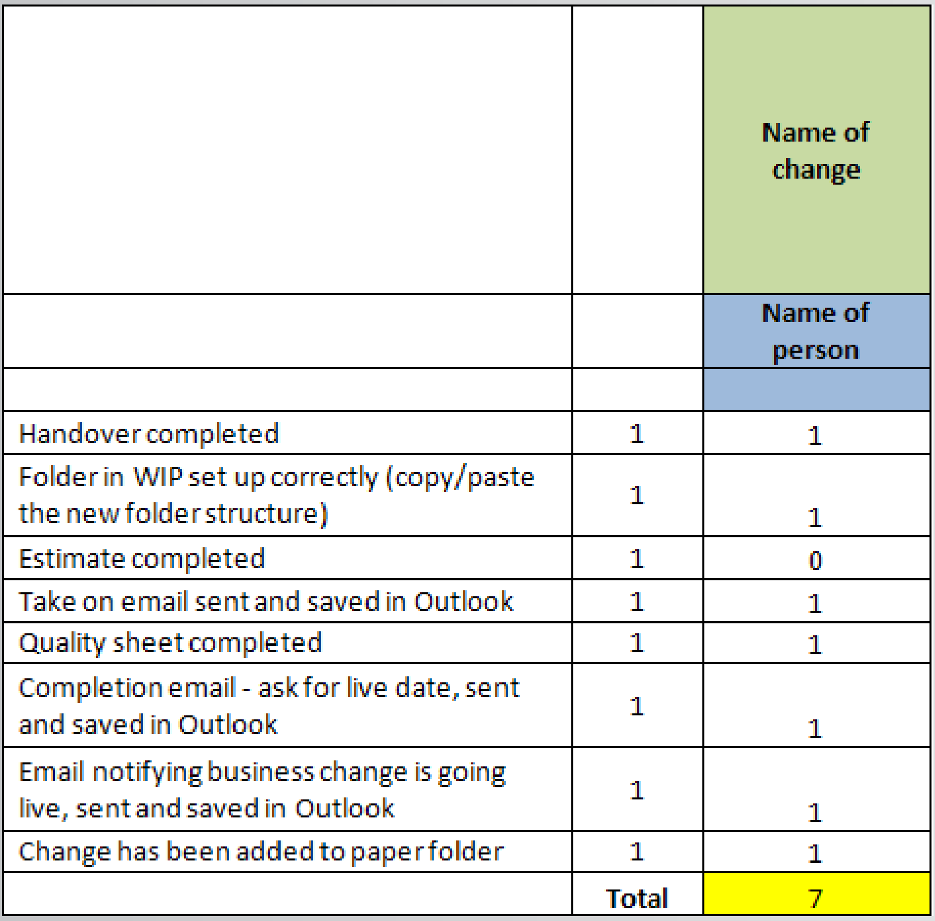
The estimate measure as if we doing things when we say we will. It is great having excellent quality, but delivering this at a rate of one change a year is not, I used this measure to provide a throughput check to quality. If a worker was consistently out on estimation, the reasons why this is the case need to be examined, it may not necessarily mean that the worker wasn't able to complete their work on time.

Reasons for an estimate being out might be:

- Inaccurate specifications causing issues that are raised at test time
- Highlighting a training issue with a particular person
- Highlighting difficulties a person had with a particular change
- etc

When looking at estimates, take caution, if the required estimate accuracy imposed is too draconian, people will quickly learn to give an estimate they can work to and we have must remember that an estimate

IMPORTANT



		Name of change
		Name of person
Handover completed	1	1
Folder in WIP set up correctly (copy/paste the new folder structure)	1	1
Estimate completed	1	0
Take on email sent and saved in Outlook	1	1
Quality sheet completed	1	1
Completion email - ask for live date, sent and saved in Outlook	1	1
Email notifying business change is going live, sent and saved in Outlook	1	1
Change has been added to paper folder	1	1
	Total	7

Fig. 1. *The process measure*

is exactly that. Ideally, use the estimation measure to encourage not discourage people from being honest in the hours worked on a change compared to the hours that they said the change would take, only in this way can the measure have any significance. Use this measure to empower the worker to critique their own work and gain valuable insight into what is actually going wrong. I introduced test exception reports as a direct result of this measure and fed these back to the development team managers to explain why a change had taken the time it had. Over the longer term, these exception reports would be examined for common themes.

The actual implementation of the estimate measure involved us breaking down the testing of a change into the categories of:

- Investigation and recreation
- Test plan creation
- Data creation
- Testing
- Second iteration
- Third iteration
- Regression
- UAT

An estimate was provided by the worker for the change in each of these categories. These were then piped into Microsoft Project. We used proprietary time logging software to log and report on time spent in each of those categories then piped that data into the project. After this I ran a using the Baseline Work Report to pull

IMPORTANT

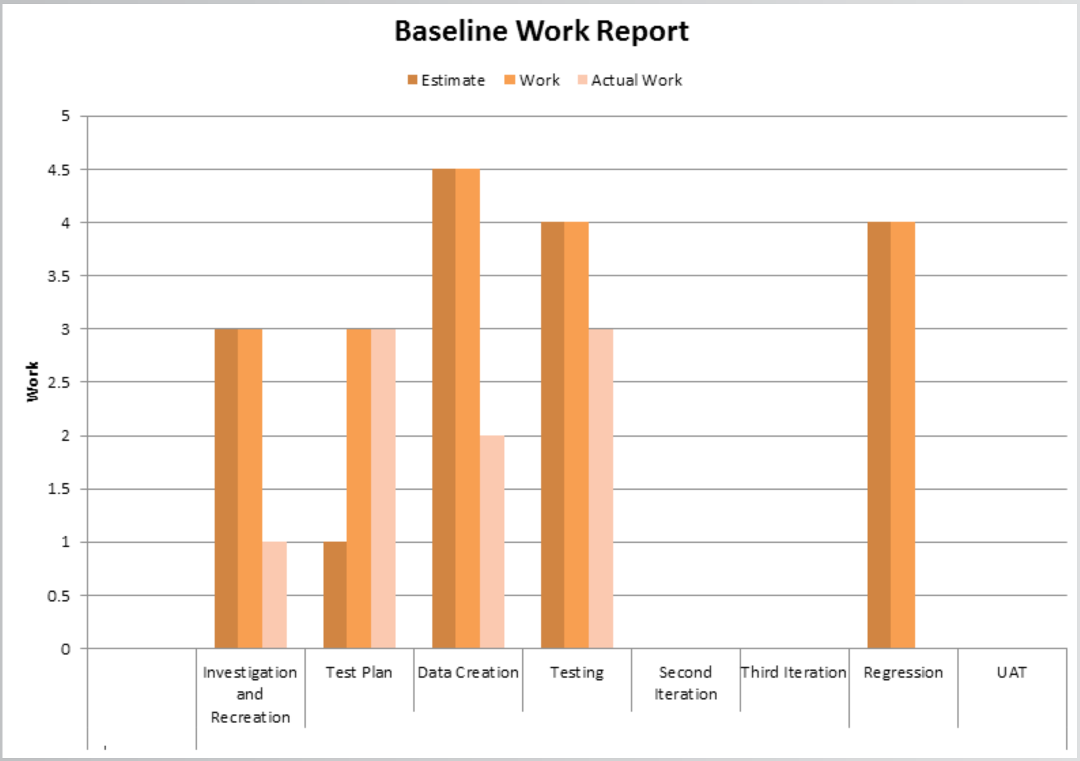


Fig. 2. *MS Project - Baseline Work Report – by change*

the data off. A negative or positive variance was calculated based on the amount of time over or under the estimate that a person was. This data per change was then rolled into an ongoing graphical representation of estimate accuracy by user and further amalgamated into a Alpha category. In this way I was able to not focus on estimates for specific changes, but whether overall that they were being hit or not, then using this data I could drill down into the categories to see where the estimate had been blown and start to ask reasonable questions about why this could be the case. Figure 2 shows the initial output from the completed change and figure 3 shows how I amalgamated this over numerous changes per user.

3) Bugs in live

We have already established that any attempt to measure quality results in a method of measurement that can be considered an imprecise science.

Bugs in live ultimately affect end users so we must include them in some kind of quality measure. A scoring model was created between cosmetic bugs (less severe) to bugs that impacted the customer directly (very severe), with bugs that affected workers (not customers) only somewhere in between.

A warranty period for changes put live already exists so when changes are put live, if no bugs are found within 90 days of that live date, the change is considered to be a success. All changes start as scoring 100% with varying percentages deducted (as determined by the scoring model when bugs are found). As bugs accumulate through the change over that 90 day period, the score for the change gets closer to 0%. Over time, this provides a good sight of how effective an individual is at testing changes (Figure 4) and this can be amalgamated further for departmental statistics (Figure 5).

The long term trends of bugs by worker are valuable here but only when looked at with caution, there may be a short term blip in someone's stats and care must be taken to look at the reasons why before any trends are implied.

IMPORTANT

Person A	
A - 0 - 10%	2
B - 11-20%	0
C - 21 - 30%	0
D - 31 % +	1
Total	3

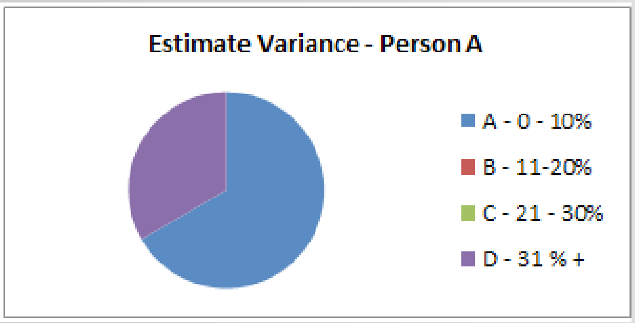


Fig. 3. *Estimates for changes completed - amalgamated by person*

Description	Score		
	Jan	Feb	Mar
Change 1	100	100	100
Change 2	100	100	100
Change 3	100	100	100
Change 4	100	100	100
Change 5	100	100	100
Change 6	100	100	100
Change 7	96	96	96
Change 8	100	100	100
Change 9		100	100
Change 10		100	100
Change 11		100	100
Change 12			100
Change 13			100
Change 14			100
Change 15			100
Number of entries	8	11	15
Average score	99.50	99.64	99.73

Fig. 4. *How bugs against changes are recorded*

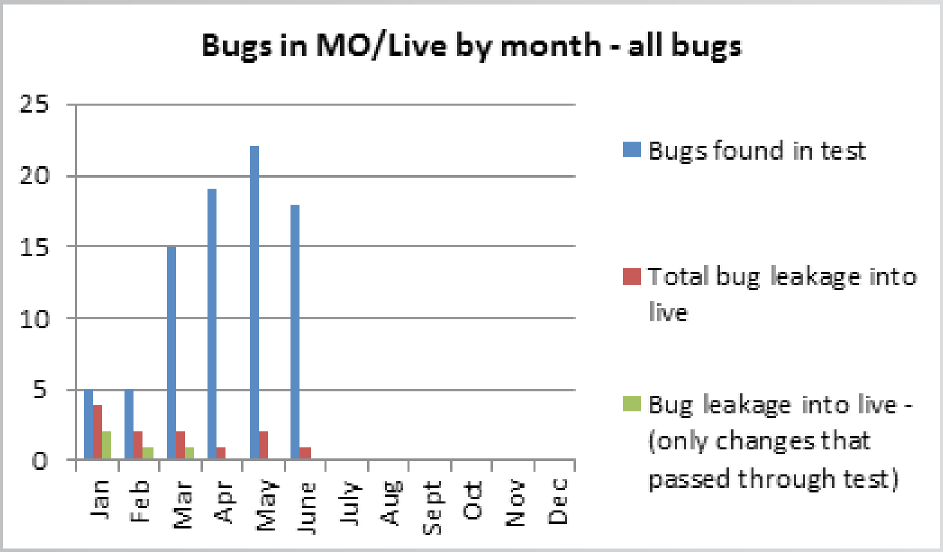


Fig. 4. *An example with hypothetical stats for how the bugs were collated*

Trends of poor quality may be caused by:

- The types of changes that person has been working on and their relative levels of complexity
- The experience level of the tester
- Difficulty in writing the change experienced by the programmer.
- Complexity of the programs altered
- etc

How would I use this information?

I chose to put all this information above into a monthly Key Performance Indicator pack for each individual, and produced an amalgamated version of this for upper management. It was very tempting to incentivise works based on quality and throughput etc, but those ideas are not without their pitfalls and it's something that you must decide for yourselves.

Summary

It is well documented that as something is measured, it often improves, but does it really? Or do we as humans become more adept at working to the boundaries we are given perhaps? Consideration needs to be given to changing measures regularly to avoid this. The problem of course is that we move into an area where new measures cannot be compared to old measures, as they say, "you pays yer money and you takes your choice". Often, the trickiest part of measuring something is not only knowing what to measure, but when you have measured it enough.

AUTHOR

Philip Young

Phil Young is a Graduate from Coventry University and is currently studying a second Degree at Swansea University whilst working for Admiral Insurance, Cardiff, UK. Phil has worked for Admiral for 15 years; starting in the call centre environment. Phil eventually earned his spot in the software testing team after helping them out with a few projects on a UAT basis. After moving to the Software Test Team and operating in a Lead capacity for a number of years, Phil was officially made a Lead Test Analyst for the Test Team "Rapid Response" and was responsible for overseeing delivery of small and valuable changes. Phil graduated to Manager of this team within a year and was an active contributor to a local Software Testing Group whose aim was to expose testers of all levels to industry differences in Test approaches. Personal commitments meant that Phil had to step down from the management role but he still continues to work as a Senior Test Analyst for Admiral.

Phil can be contacted at philyoung99@googlemail.com

Discover

the different faces of

Testing

@

Belgium Testing Days

March 17-20, Bruges

Discover the Belgium Testing Days Conference – the innovative software testing "**doing conference**" that brings you hands-on workshops, interactive discussions sessions, advanced peer inter-communication, an innovative lab with the most up-to-date knowledge, tools, and technologies available in the industry today. The Belgium Testing Days is at its 5th edition. Meet **industry experts**, colleagues and peers in an international test and QA community for 4 **days** jam-packed with **learning** & **doing sessions** that will help you make a powerful impact **in your job role** and for **your company**.

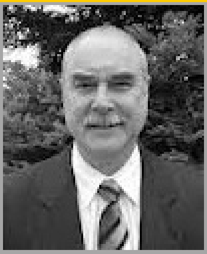
FACTS

1. 12 PRE-CONFERENCE WORKSHOPS
2. +58 SPEAKERS WITH INTERNATIONAL FAME
3. 5 KEYNOTES, INCL. 1 OPEN PANEL DISCUSSION
4. 5 ADVANCED WORKSHOPS
5. A INNOVATIVE LAB FOR DEV'ERS & TEST'ERS
6. TOPICS & CASE STUDIES COVERING OUR ACTUAL DILEMMA'S, ISSUES & CHALLENGES
7. INTENSE NETWORKING & INTERESTING SPONSORS

Visit us www.btdconf.com

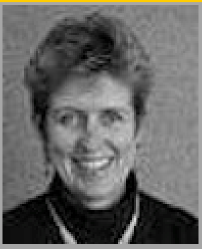
Keynotes & Workshops by international famous knowledgeable colleagues

Dorothy Graham


Doug Hoffman

Fiona Charles

Stuart Read


Lisa Crispin

J.J. Cannegieter

Rob Sabourin

And Jerry E. Durant, Goranka Bjedov, Seretta Gamba, Isabel Evans, Derk -Jan De Grood & many more...

QUALE

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
Piotr Poznański
piotr.poznanski@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

Architecture Blue Building Business City http://pixabay.com/en/architecture-blue-building-business-22231/

Other

Laser Laser Light Research Laboratory Science http://pixabay.com/en/laser-laser-light-research-11646/

Pen Office Leave Office Accessories Lid http://pixabay.com/en/pen-office-leave-office-accessories-62374/

Newton's Cradle Balls Sphere Action Reaction http://pixabay.com/en/newton-s-cradle-balls-sphere-action-256213/