CORE

CORE Nº 1 - October 2010

Engineering Productivity:

Some ways to measure it and manage it by Tom Gilb

Agile software development with distributed teams

by Jutta Eckstein

Product Qualities Approach

by Ryan Shriver

Efficient Testing with All-Pairs

by Bernie Berger

Global Association for Software Quality

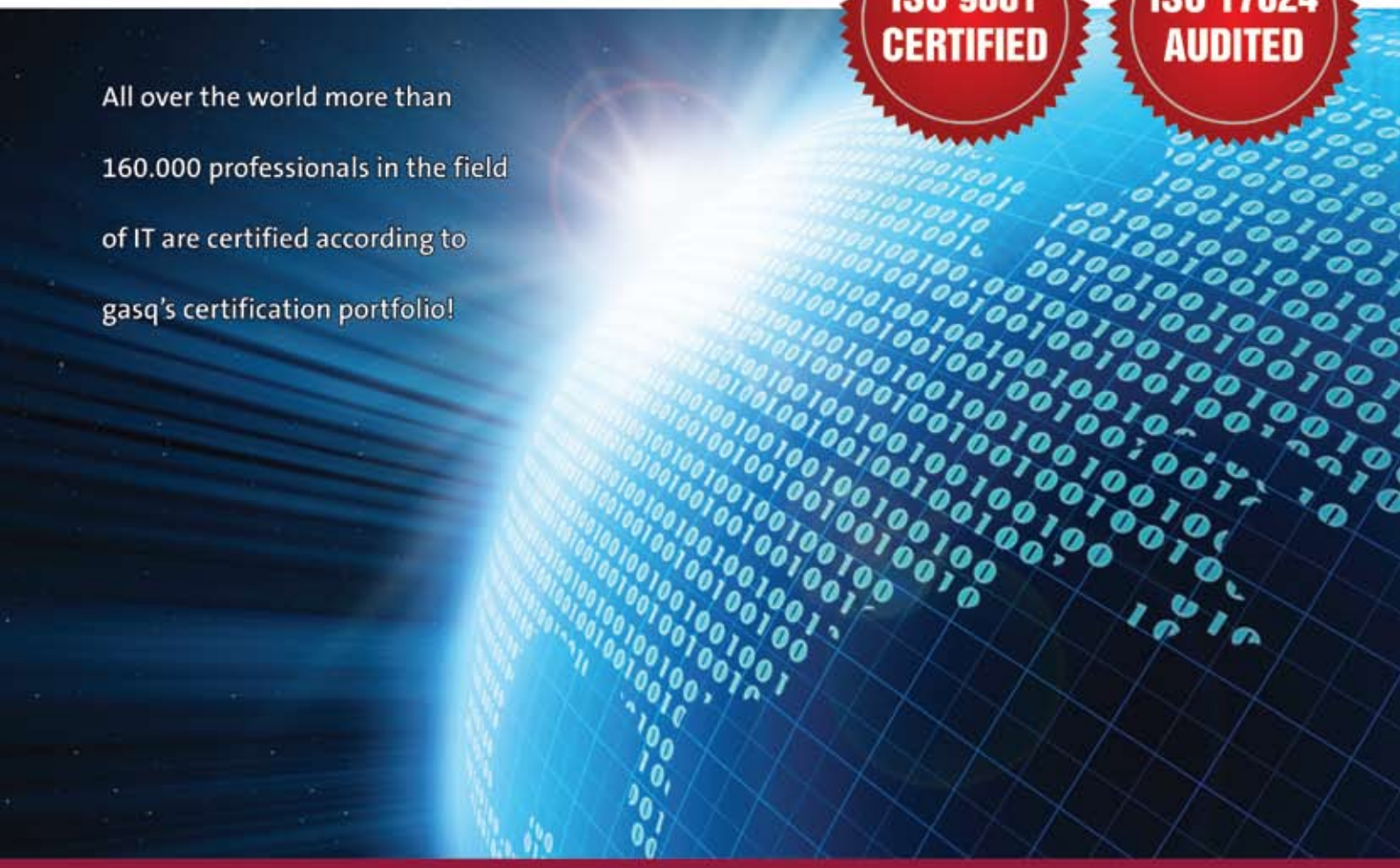
gasq – the global Certification Provider

gasq – the Global Association for Software Quality – is one of the leading Certification Provider for IT-personnel certifications.

ISO 9001 CERTIFIED

ISO 17024 AUDITED

All over the world more than 160.000 professionals in the field of IT are certified according to gasq's certification portfolio!



www.gasq.org

INDEX

Welcome

Management

Quality in Project

Software Engineering

Software Testing

Editor's note

Quarterly **c0re** (4 numbers per year) is published by gasq Service GmbH.

WWW
www.coremag.eu

Chief editor:
Karolina Zmitrowicz
karolina.zmitrowicz@coremag.eu

Editorial Staff:
Bartłomiej Prędki
bartlomiej.predki@coremag.eu

Mailing address:
C0RE Magazine
c/o gasq Service GmbH
Kronacher Straße 41
96052 Bamberg
Germany

Advertisements:
info@coremag.eu

Persons interested in writing are asked to contact:
editor@coremag.eu

About us

Bartłomiej Prędki

Started his professional experience in 2004 as a tester of mass-market mobile applications. Within next years he gained experience in Testing and Quality Assurance areas, mostly focused on Telecommunications industry. During his career he was involved in testing, managing testing processes, training, technical support, requirement analysis, recruitment, technical documentation creation and review.

Besides his mobile and telecommunications experience, he was also involved in financial and banking system projects.

Holder of ISTQB Advanced Technical Test Analyst certificate. He lives and works in Wroclaw, Poland.

Karolina Zmitrowicz

Karolina started her professional experience in 2006 as a Quality Assurance specialist working on international projects in banking sector. Within next 6 months she become Test Leader and Lead QA Consultant supporting testing activities on customer side and leading customer test team.

During her career she worked as a Test Leader, Change Manager, Technical Writer, Business and System Analyst. She has been involved in testing and managing testing, requirement engineering, business analysis, trainings, technical documentation creation. She currently works as a System Analyst for leading Polish insurance institution.

She is an author of several publications in QA and BA area, trying to apply best quality assurance practices in analysis and requirement processes.

Certified ISTQB Test Manager and REQB Certified Professional for Requirements Engineering. Certified quality management systems manager and leading auditor (ISO 9001: 2008).

Currently she lives and works in Warsaw, Poland.

Welcome to the c0re magazine

"Quality is not an abstraction; it's a measurable, manageable business issue."

John Guaspari

What does a quality mean for testers? What does a quality mean for business analysts? What is a quality for developers? How quality is perceived by project managers?

How can you ensure the quality of a product? How to verify it?

What techniques can be used to improve the quality of a process or product? What kind of tools can be used?

With your help, we will try to answer these questions.

We would like to introduce **c0re** magazine – free magazine focused on quality in IT. Our aim is to provide global platform for IT professionals to share knowledge and experience in quality area. We would like to present different points of view and different perspectives on quality. If you are interested in quality assurance, testing, process' quality – this magazine is for you.

We know many websites, blogs, forums concerned with QA and quality. We know many authors writing interesting and valuable papers – unfortunately they are not known to broader community – we want to introduce them.

Why not to create a tool for exchanging experience and information, which people working in quality area can share in one place? Why not to help people who have just started their QA career to gather knowledge and learn from the best international experts?

Based on the above questions we have created the concept of c0re – magazine created by specialists for specialists.
In **c0re** magazine you will find articles written by international experts, grouped in four basic sections:

• Software engineering - topics which cover requirements collecting and analysis, designing, software development life cycle, development methods, change management, configuration management etc.

• Software testing - practical aspects related to testing - techniques, methods, tools.

• Quality in project - quality assurance and control in the process level.

• Management - for those, who are involved in QA team leading, project and process management.

In addition, some other sections exist as well: book's reviews, notifications about events and conferences, tools' reviews and comparisons, tips & tricks, feuilletons.

We hope the magazine will meet your expectations and give a motivation to further self-development. We hope that some of you will decide to share your experience and present it to our community. Your insight will help us to shape the c0re according to your needs - providing comments on the contents of the magazine, providing new ideas and proposals. We invite you to cooperate with us!

Chief editor
Karolina Zmitrowicz

Agile software development with distributed teams

intermediate

Author: **Jutta Eckstein**

About the author:

Jutta Eckstein is an independent coach, consultant and trainer from Braunschweig, Germany. Her know-how in agile processes is based on over ten years experience in developing object-oriented applications.
She has helped many teams and organizations all over the world to make the transition to an agile approach. She has a unique experience in applying agile processes within medium-sized to large distributed mission-critical projects. This is also the topic of her books 'Agile Software Development in the Large' and 'Agile Software Development with Distributed Teams'. She is a member of the AgileAlliance and a member of the program committee of many different European and American conferences in the area of agile development, object-orientation and patterns.

Understanding Agility

A historic marker indicating that agile methods no longer would be considered mere hype or a fringe movement was Scott Adams' Dilbert comic strip on agility[1] . With every passing year, agile concepts have become more firmly ent renched in mainstream business and, today, are largely accepted in the modern market. Of course, while noting the movement of agile methods from the realm of fringe, Adams also exposes typical misunderstandings, illformed expectations, and downright strange interpretations that some think still pervade the agile approach.

Agility has come into its own as a value system defined by the Agile Manifesto[2] . Based on twelve principles created to ensure the value system[3] , the Agile Manifesto demonstrates that there is more to agile development than just one specific methodology, such as Extreme Programming[4] or Scrum[5] .

The first value stated in the manifesto favors "individuals and interactions over processes and tools." The processes referenced in this first value statement include agile development processes, which means teams must ensure that their development process supports their needs in the best way possible. Using the principles in the manifesto, teams can find guidance on how to modify and adjust their development processes to best support their needs.

Core Value Pair Statements

The values expressed in the Agile Manifesto apply to all agile projects, superceding guidelines of any specific agile process. The core of the manifesto compares in four statements two values and argues that although each value provides a value in general, the first value is more important than the second and that the latter half of the each statement is only valid if it supports the former.

• Value Pair Statement #1, "Individuals and interactions over processes and tools," highlights the idea that it is always the people involved in a project and how they collaborate that determine a project's success or failure. The manifesto does not devalue processes and tools (otherwise, we wouldn't talk about processes, and the agile community wouldn't have created tools such as unit-testing frameworks, integration and configuration management tools, and others), but if individuals don't work together as a team, the best tools and processes won't help the project succeed.

• Value Pair Statement #2, "Working software over comprehensive documentation," is perhaps the most often misunderstood of the four statements. People unfamiliar with agile development may mistakenly believe agile projects don't document, or even disdain documentation. Not so. In the same way that processes and tools play a major role in successful development, documentation also plays a major role. However, this value comparison expresses that working software is the critical success factor for any development effort. Documentation might be needed to support or to understand the working software, but it can't and shouldn't be an end in itself.

• Value Pair Statement #3, "Customer collaboration over contract negotiation," emphasizes that although you need a contract, it can never be a substitute for a good relationship with your customer. In order to deliver a satisfactory product,

involve customers regularly throughout the development process.

• Value Pair Statement #4, "Responding to change over following a plan," advocates the importance of reacting to changes (especially in terms of requirements changes), rather than sticking to an inappropriate or obsolete plan. We accept that both the customer and the project team will learn over time, and we want to acknowledge this learning and incorporate it into the development effort. If the finished product delivers what the customer and we planned for before confronting changes and disregards anything we learned during development, the product will be a failure, even if it fulfills a contract.

The agile value system accommodates collocation as well as distributed software development. Later in this chapter, I examine implications of agile principles regarding globally distributed projects.

Systemic Approach

Agile development promotes a systemic approach that is supported by a closed-loop routine of planning, doing (or performing), inspecting (or analyzing), and adapting, as follows:

• In Planning, plan immediate activities (having broken down a development project into deliverable chunks, begin planning for the first task). This is most often short-term planning, focusing on the next iteration, but it can also be long term, such as planning the next release.

• In Doing, perform activities planned in the first step.

• In Inspecting, analyze the performance of the activities planned in the planning step. Did all work as planned? Was there a specific process that worked well and would be appropriate to repeat in the future? Did a specific process or plan fail or require adjustments for the future?

• In Adapting, determine what kinds of adjustments the previous inspection step revealed are needed in order to improve development. In this step, decide necessary actions for the following iteration.

The last step in this closed-loop routine provides input for the first step in the next round, and so on.

Risk Reduction

The goal of an agile project is not only to deliver a product at the end of the project's lifetime (called a deadline), but as well to deliver early and regularly. In order to do so, we divide the project's lifetime into development cycles. A bigger cycle that produces much functionality (sometimes called a feature pack) is called a release. Within that, we use a smaller cycle to organize work in smaller chunks, and to deliver smaller functionalities. This smaller cycle is called an iteration[6]. Both a release and an iteration lead to a delivery or a potentially shippable product.

A tremendous advantage of agile development is risk reduction through high visibility and transparency. By developing iterations of a working system, receiving regular feedback from the customer and from tests, and with tangible progress, you have access to the actual status of the project. Knowing the actual status of the project in turn enables you to make decisions regarding further deliverables and necessary actions. For example, if you encounter that the system does not fully satisfy the customer and it can't be turned in the right direction, you have the advantage of being able to stop the project early, before all the money has been spent.

The Productivity Myth

Another common, and misguided, argument is that following an agile approach will greatly increase a development team's productivity compared to other approaches. While this can be true, it is not always necessarily so. Agile development guides a team to deliver a working system frequently—"frequently" meaning in iterations lasting one to four weeks. A "working system," on the other hand, is defined by the customer's evaluation of usability. Thus, by providing a working, usable system periodically, say, every two weeks, an agile team ensures maximum business value for its customer.

Therefore, following this approach, your customer might decide to proceed with an operational system earlier. This will give your customer a market advantage. However, it does not necessarily mean that the project as a whole is finished—meaning all required features are implemented—earlier.

More Than Practices

Agility is more than a collection of practices. Every so often, I hear people mixing up specific practices with agility. Practices—for example, Extreme Programming's pair programming or test-

driven development—are a great means to preserve the agile value system; however, these practices are not the value system itself. For instance, you can successfully apply pair programming and use a linear (or waterfall) development approach.

Neither Chaotic nor Undisciplined

Many people consider the agile approach to be an undisciplined approach. Some regard agile as an ad-hoc approach that doesn't require any planning, one in which people act independently according to whim. Sometimes, the agile label is used as an excuse for lack of preparation. For example, if a person has to conduct a workshop or deliver a talk and doesn't prepare material, his or her presentation will consequently follow an ad-hoc approach. This person might argue that the approach used is agile, and therefore doesn't require preparation or planning. Instead, absolutely the opposite is true: Agility requires a lot of planning, even more planning than a linear approach. As Lise Hvatum states, "Agile is highly disciplined and more difficult, requires more maturity, than waterfall[7] ."

The reality is, agile requires and embraces planning. In agile development, the artifact of a plan is not overly important; the activity of planning, however, is essential. Jakobsen contrasts a choice between an old management style—for example, Taylorism, where managers dictate procedure—and an innovative management style—such as Lean Jidoka[8] , based on trust, respect, empowerment, and belief that it is the people who use a process who are best able to improve it[9] .

Improving processes means changing your original plan, and preparing for future re-planning to utilize what you learn as development occurs.

1 See S. Adams, Dilbert (http://www.dilbert.com).

---

2 See the Agile Manifesto online: http://agilemanifesto.org. For an analysis of the Agile Manifesto, see A. Cockburn's Agile Software Development: The Cooperative Game, 2nd ed. (Boston: Addison-Wesley, 2006).

---

3 For my thoughts on agile development for large projects, see Agile Software Development in the Large: Diving Into the Deep (New York: Dorset House Publishing, 2004).

---

4 For Extreme Programming, see http://c2.com/cgi/wiki?ExtremeProgramming Roadmap.

---

5 For Scrum, see http://www.controlchaos.com and http://www.scrumalliance.org

---

6 In Scrum, an "iteration" is called a "sprint." I personally do not like that term because, for me, it connotes frantic, unreserved effort. Iterations should involve adequate resources so that teams are not racing to finish.

---

7 L.B. Hvatum, personal communication.

---

8 Lean Jidoka requires all team members to be responsible for improving the process (immediately) as soon as the quality of the outcome decreases.

---

9 C.R. Jakobsen, personal communication. ■

How to be a better tester

basic

Author: ***Jan Sabak***

About the author:

Jan is a software quality assurance expert. For fifteen years he has been working on testing and quality of software and hardware. He holds MSc in Computer Science of Computer Science Department at Warsaw University of Technology. He built and managed Quality Assurance Departments in Matrix.pl, IMPAQ and Infovide. Currently he works on his own consulting company (AmberTeam) which strives to assure peaceful sleep of CIOs and project managers through risk measurement and management. He is an active promoter of the knowledge and culture of the quality of software development. He is a President of the Revision Board of SJSI (Association for the Quality of Information Systems). He possesses ISTQB CTFL and CTAL TM certificates.

Abstract

I would like to give some thoughts about what it means to be better tester. Who can and should be a better tester? Why someone may want to be still better in his/her job. I will concentrate on personal development as opposed to training courses. There are some traits a tester should posses and they cannot be taught on courses. How to acquire and develop those traits?

Companies always need better people, especially now in the days of staggering economy. Is it possible to motivate people for personal growth? And if it is how to do that? I would like to show the way organizations can enable and facilitate personal growth of testers to the mutual benefit of employers and employees.

At each level of process maturity the goals are different. The measurements we collect to understand the status of maturity level goals achieved is explained in this paper using GQM model.

Introduction - Traits of a tester

To be a good tester and to perform testing well may mean several different things. There are many different tasks during testing, and these tasks may require sometimes quite different knowledge. In this article I will concentrate on the role of a tester as a person responsible for analyzing test basis, designing test cases, implementing them, performing and reporting test results. I will not elaborate on managerial tasks because they are more complicated, but some conclusions can be generalized basing on this material.

Skills and knowledge which testers need can be divided into four categories. They are:

- knowledge about testing process and techniques
- technical knowledge
- domain knowledge
- personal traits

Much of it can be learned through training courses (e.g. testing process, some of domain knowledge), but some of it comes with experience during performing tests in projects. Still, some of being a good tester requires possessing certain personal qualities.

ISQTB Foundation Level syllabi list following traits of a good tester:

- curiosity
- professional pessimism
- critical eye
- attention to detail
- good communication with development peers
- experience on which to base error guessing

One may argue which is more important, hard testing knowledge or soft testing skills. Some test managers believe they can get anyone with potential of being a tester (i.e. having a personality of a tester as listed above) and make him or her a good tester.

I will not give the answer to this question here as this paper is addressed to these who are already testers, and these who want to be better at their job, and maybe to those of us that do not feel the need to go forward and become better and better.

Why to be a better tester?

There is a saying that who does not go forward falls back. The systems we are testing and technologies we use are day by day more complex. Computer science is in its development stage yet, so each year new techniques, methodologies and tools emerge. Testing is a set of activities that is a part of more general project development. Test planning always tries to match testing tasks to project management and production methodologies (see V-model and W-model). If project design and development methods go forward, testing methodologies must proceed in parallel.

Moreover, organizations we work

for are developing. Our competition is developing as well. If we allowed ourselves to do things the way we always did, others would outperform us having lower prices and tighter schedules.

These two above mentioned factors are external to the testers. But some of testers' personal traits (e.g. curiosity and critical eye) constantly motivate them to learn and develop, to reach for more knowledge.
Testing has been gaining more and more visibility over last few years. Testing community has developed new standards and aids for testers (e.g. ISTQB Syllabi, ISO/IEC 29119). There are new tools that support testers, but testers have to get to know them, try them and learn their usefulness. This is also a form of personal development

There are many reasons for testers to develop. Some of them are external to testers and some of them come from the essence of being a tester. Both are important to drive tester's career forward. And where is the will there is the way. There are many ways to get knowledge and develop skills. In next chapter I examine some of them.

Development of a tester

As I stated in previous chapter all testers need to improve themselves, need to learn new techniques and tools. There are many ways to do that. Most common of them are:

• training courses

• conferences

• self development through books and articles

• hands on experience in projects

Training courses can give you knowledge and insight into experience of an expert who givies a course. Training courses may prepare you for an examination giving certificates. Training courses in the form of workshops help building up skills in using tools and techniques. Some skills such as assertiveness can be learned effectively only through workshops.

Equally important are other forms of development such as conferences that create opportunities to exchange experience and share problems and their solutions.

Every organization should have personal development plans to keep balance between project work and self development. It is hard to tell how much of personal time should be devoted to development. The more work in projects is done the greater profit it brings to organizations in the short run. But in the long run too little development can cause demotivation and render work methods obsolete. If there are no plans and no test managers to keep an eye on their realization, all time and attention is devoted to current projects and none of them to development. This brings stagnation in well known comfortable procedures and boredom and after a while best people, who value their careers begin to seek opportunities for growth elsewhere.

While building development plans and assigning development goals to testers one has to keep in mind that they have to be formulated in the right way. For example using SMART technique:

• specific

• measurable

• ambitious

• realizable

• timely

Very good technique for defining goals can be found also in the book "One Minute Manager" [3].

Most important factor in giving work and development tasks is their difficulty. Too easy tasks are boring and do not really bring any development. On the other hand too difficult tasks frustrate, demotivate and do not improve skills or knowledge as well.

People have three zones of

competence:

1. comfort zone

2. development zone (discomfort zone)

3. panic zone

If a tester gets tasks only from comfort zone, he or she can execute them well and on time. But this work becomes boring with time and does not improve him or her. Furthermore boring work may be done without due attention and a tester may overlook defects in tested software.

Too difficult tasks may induce panic in some people that may inhibit them and again no development will be achieved or work tasks will be performed inadequately. That may raise project's exposure to risk.

To maintain tester's curiosity, attention to detail and other desirable traits, to develop testers, they have to be given tasks and goals that are in their discomfort zone. These tasks should require more than a person thinks is safe and easy but not too much so as not to upset a tester.

If you are a tester living and working in comfort zone, notice that this may be easy and pleasurable, but it is not developing. To develop you need to leave your comfort zone and try things out of its bounds. See also [5] for some motivation to do so.

Deliberate practice

In order to become a better tester it is not sufficient to do one's work in the best way. To grow one must have a goal and a plan to achieve it, and of course goal has to be taken from discomfort zone. Everyday exercise complementing more formal training can help personal development and can give new meaning to routine tasks.

In his book [4] Geoff Colvin describes deliberate practice, an everyday exercise which aims at constant personal growth. It requires dedication and patience but assures that person who devotes time to it will develop desired skills or traits.
Looking at chess masters, musicians and sportsmen we can learn to practice our skills too. Not every exercise brings development. Such practice has to possess a couple of important features:

• one has to intentionally strain to develop oneself

• one has to exercise every day

• one has to regard it as most important
• it has to be hard

• it does not need to be pleasurable

There is no way to motivate people to work hard to develop. An impulse for doing that has to come from inner motivation. The role of a manager of people wishing to grow is to help them doing it within organization and if necessary to be a mentor to such people.

Deliberate practice can be present in organizations but it is hard to achieve because quarterly or yearly appraisal rituals effectively hide actual problems of a worker and concentrate on filling in the forms and tactical placement in corporate ladders and salary brackets.

To implement everyday development practice organizations need to notice that the best way to raise engagement in development is to inspire and not to command. To do so organizations have to develop a corporate culture that allows and promotes personal development. Organizations should choose some of their employees and allow them to be mentors to others. To share experience and to judge grow of their mentees.

Even if we do not have mentors we can exercise personal growth through deliberate practice. Having mentors gives, however, better results because a mentor can help to choose appropriate goals and can evaluate outcome of exercises and show their weak points.

To do deliberate practice at work one has to take care of several crucial elements. It has to be prepared, observed and measured and conclusions have to be drawn from it.

Before work

Before starting working day you have first to examine your beliefs. You have to believe that work and practice is beneficial to you. That it helps you grow and develop you career, to raise the level of your expertise. You have also to believe in you ability to perform the work that lies ahead of you, believe in your self-efficacy.

Before starting work you have to set a goal for today. The goal has to be SMART, and it has to lead you out of your comfort zone into your development zone, but not too far away into panic zone. Remember that lazy people do not set goals and live in their comfort zone. Mediocre people set goals of immediate results. And best people set goals of personal improvement or improvement of work methods and processes.
After you have decided what today's goal will be, make a plan of achieving it through today's tasks.

If you have a mentor, he or she may help you to choose your goals. He may be able to see the long term goal you try to achieve and can align your goals with needs of the project or organization. He has already walked the path you are taking and may be able to point out the best next step for you.

During work

During executing work tasks you have to observe yourself and methods you use, your ways of thinking. This is called metacognition and is a way of thinking about thinking. This allows you to correct and rearrange the way you are working in order to achieve the goal you have set in the morning.

If you have a mentor, you may work with him or her. It is easier to observe someone else than to observe oneself. From his experience he may also see more good points of your work or things you need to improve.

After work

After work your mentor or yourself appraises your work. You should measure to what extent you goal has been achieved. Appraisal and measurement should be as accurate as possible. You have to get to know what you have done well and what you have done poorly. The key to get better is to know you deficiencies and to repeat situations that showed them to invent and drill better ways of dealing with those situations.

Important factor here is to acknowledge that all defeat has its source in you yourself. Admitting that allows you to set more realistic goals and to take responsibility of you development.

Above procedure allows anybody to simultaneously perform work tasks and develop qualities that he or she needs. Thanks to deliberate practice even most boring job gets additional meaning and a greater goal. This raises motivation to work and to continue self development.

Deliberate practice at tester's work

Deliberate practice can be used during tester's work. Many task in software testing produce measurable outcomes. Many processes have corresponding standards that can be used to judge tester's performance, e.g. ISO 29119 or IEEE 829.

To be a better tester one has to get more of what has been mentioned in first chapter:

• knowledge about testing process and techniques

• technical knowledge

• domain knowledge

• personal traits

Through deliberate practice you can improve skills or get knowledge from any of above listed groups. If you need technical knowledge and you are implementing automated test scripts you may set a goal of employing into your tests today one new feature of automated testing tool which you are using. In such case you should choose an area in which the feature may be useful, get to know the feature in details and use it as appropriate. After the work you may have the tests you produced reviewed by more experienced test automation engineer and he or she will tell you what you did well and what aspects you should improve.

If you need skills in using testing techniques and you plan today to design test cases, you may choose one testing technique and try to design more test cases to each test condition, even if you have already covered them. At the end of the day you will be able to use that technique far better and also you will be able to tell for which types of testing conditions it should be applied and how. If you need more inspiration in setting everyday goals of personal development you can refer to ISTQB syllabi. Both Foundation Level and Advanced level syllabi contain learning objectives. Learning objectives are divided into four levels:

• K1 – remember

• K2 – understand

• K3 – apply

• K4 – analyze

They are also structured by the chapters of syllabi. And for example if your general development goal is to more effectively use reviews you may refer to chapter 3 Static techniques of Foundation Level syllabus and chose one of learning objectives from that chapter. For example "Recognize software work products that can be examined by the different static techniques". You can make then a checklist for your project to use in project planning, which will help checking if all necessary reviews have been planned.

If you need to work on some of you personal qualities you may choose one of traits of good tester and make a goal for the present day of it. For example if you want to cultivate professional pessimism, you may try to think about project and product risks in tasks you are working on. Make a goal of performing twenty mini risk analyses today and put a dash on a sheet of paper for each analysis and note how many risk items you were able to think of.

These are examples only of many different directions and goals you may choose from while planning personal development of you or your testers. If you add to that pieces of domain knowledge testers need to possess in order to design and run tests in projects, you get a vast number of opportunities to learn.

Summary

Every tester needs to learn and to polish his skills. There are many ways of doing that. First of all you need to achieve level of so called conscious incompetence. That drives learning and training courses can carry you from conscious incompetence to the level of conscious competence. In that level you are able to perform you work, but you simultaneously think of the way you are doing it. This level is still in your discomfort zone. As your experience grows, you are growing more and more accustomed to using skills you acquired during training course or workshop. These skills move more and more into your zone of competence or more precisely you competence zone expands to include those new skills and techniques. When your work hides completely into comfort zone your personal development stops. And at that moment you need deliberate practice which will move you back into conscious competence level and out of comfort zone and allow you to grow.

In this article I showed the need of personal development of tester from the point of view of a tester himself and an organization he works for. I showed several ways of acquiring new skills and knowledge. And I described deliberate practice and its application which can help sustain personal growth on daily basis, in which all testers should participate.

Reference

1. International Software Testing Qualifications Board: Certified Tester Foundation Level Syllabus

---

2. International Software Testing Qualifications Board: Certified Tester Advanced Level Syllabus

---

3. Blanchard K. H., Johnson S.: One Minute Manager, William Morrow; Later Printing edition (September 1, 1982), ISBN 978-0688014292

---

4. Colvin G.: Talent Is Overrated: What Really Separates World-Class Performers from Everybody Else, Portfolio Hardcover; 1 edition (October 16, 2008), ISBN 978-1591842248

---

5. Johnson S.: Who Moved My Cheese?: An Amazing Way to Deal with Change in Your Work and in Your Life, G. P. Putnam's Sons (September 8, 1998), ISBN 978-0399144462

Optimal testing tasks management using Critical Chain scheduling

intermediate

Author: ***Ladislau Szilagyi***

About the author:

Ladislau Szilagyi holds a BS in Computer Science Mathematics, 1978, Bucharest University, Romania and has more than 30 years of working experience IT. He worked until 1991 at the Research Institute for Computer Science, Bucharest, authoring multitasking real-time kernels and process control systems. He was involved in software testing since 1995 and works now at Totalsoft, Romania. His main specialized areas of consulting and training are Quality Management, Testing, Requirements Engineering and Software Process Improvement. He published several articles in the Carnegie Mellon's Software Engineering Institute Repository (https://seir.sei.cmu.edu/seir/) and several software magazines (Testing Experience, Quality Matters, What Is Testing), covering topics such as CMMI, Testing, Metrics and Theory Of Constraints. He is also a speaker at software testing conferences (SEETEST 2008 & 2009, Testwarez 2009). He is ISTQB certified (CTFL, CTAL) and an active ISTQB trainer.

TotalSoft

Contact: lszilagyi@totalsoft.ro

Abstract

The approach to project management known as "Critical Chain" provides mechanisms to identify and protect what's critical from inevitable uncertainty, and as a result, to avoid the impact of Parkinson's law at the task level while accounting for 'unpleasant surprises' at the project level.

Some tales from a real testing project

Some years ago, I was the test manager for a large software project. The project team used a RUP-like software engineering model, handling a large number of software documents, and the project manager scheduled all tasks using a Gantt chart, detailed to the level of something like 'half-day-task'.

I was hence constrained to use a similar scheduling scheme, for the testing activities, because the project manager insisted on receiving milestone reports from the team members. There were only 3 milestones, including the project delivery date.

My test team consisted of three testers, let's name them Ann, Basil and Colin. The tasks were the usual ones:

- Specification review
- Quality risk analysis
- Test cases development
- Test cases review
- Test environment setup
- Test data preparation
- Integration test sessions
- System test sessions
- Acceptance test sessions

I asked everyone to provide me estimates on their task's effort, and I used these estimates to build the testing project schedule; to be honest, I remember that I even added 20% to each estimate, as a 'safety buffer', to minimize the risk.

Then, I assigned the tasks to the testers using the classical 'critical path' method. They all had fixed task delivery dates, and everybody agreed to deliver on time. The first month passed without any problem, the team apparently was calm and relaxed, talking and joking all day long. They kept me telling 'we are in schedule'.

When the first milestone was in about two weeks, I witnessed a very strange and unexpected behavior from Ann. She suddenly changed attitude, coming to me and asked: "…hey, listen, shall I wait a week more for those damn' test cases, or what?". I was perplexed, because I knew that Basil and Colin were assigned to write all the test cases and hand them over to Ann for review, and they newer told me that something was behind schedule. I quickly improvised a team meeting and learned with surprise that in the last week all three were …waiting. Basil and Colin practically finished one week ago the test cases, Ann practically finished a week ago the quality risk analysis, but they all waited until the 'delivery' date. Ann was more extrovert than Basil and Colin, so she was the first to come to me and complain, but unfortunately only after a week of doing-nothing.

A month later, I noticed from the team another apparently strange change of

behavior. They all looked very worried, stopped the usual small talking and stayed until late at work. It turned out that all three were assigned some extra-work from the CEO (of course, with the mandatory comment 'don't tell him about this…'). And, as a logical implication, all three were now behind schedule for the testing tasks.

The classical scheduling method major problems

Test managers build the schedule and fix the deadlines from estimates of duration required by the various tasks that comprise the test project by doing first a high-level estimation based on historical data, and then by asking the team members about their personal tasks estimation. There are also other effort estimation techniques, but anyway the historical data and the personal task estimation are often used. Testers know that they will be held accountable for delivering against their estimate. Therefore, it is prudent that they include not only the amount of time they expect the work to take, but also some extra-time to protect their estimation. So testing task estimates have plenty of safety in them, supplementing the

The test manager then uses these estimates and builds them into a list of dependent tasks with associated start-dates and due-dates. It's not unusual that the test manager will add an extra-buffer to the initial task duration, often expressed as a fixed percentage (let's say 10% of the initial task size). Testers plan their work around these dates and focus on delivering their deliverables by these dates [2,3].

But, in practice it often happens that a tester receive some other urgent job, regardless on his current assignment. And he has plenty of time until the promised date to finish the original work, which at this point looks like a long way off due to the safety included in the estimate. So, in the most cases, he is easily putting off or delaying the original work in favor of other stuff because the due date is out there, in a 'safe' future. This "urgent job" takes precedence until he sees the scheduled due date coming up on him. Now the originally scheduled project task is hot. He starts working hard to finish the original task on-time…but, usually, it's too late for this.

The first problem which strikes now is that he can't know what problems will impact him until he starts the work. And he started the work later than planned, after eating up most of his 'safety interval' because of the other important work. There isn't time left to recover from the problems in time to meet the due date, at least without heroics, burnout, or loss of quality because of bugs that 'escape' unnoticed. So, this way the testing task deadlines get hard to meet...and cascade through the testing project, putting the promise of the final delivery into jeopardy, which creates new "urgent stuff" which impacts other projects...and so on and so forth.

The second problem is the Parkinson law ("work always expands to fill the time available"). Even if, by some miracle, a tester will finish a testing task early, will the required 'next' tester be available to pick it up? Or will some other tester feel an urgency to pick it up? The answer is no, because everybody will strive to 'protect' its own 'safety'. So, in these circumstances, the testing project is pretty well doomed to meet the final target date at best, but in all likelihood missing it, or just making it with burnout heroics, bad testing quality or poor test coverage. There is also the so-called "Student syndrome" - many people will start to fully apply themselves to a task just at the last possible moment before a deadline. This leads to wasting any buffers built into individual task duration estimates.

This all occurs due to the combination of task due dates and realistic, prudent, "safe" estimates. We protect our testing project due dates by protecting testing

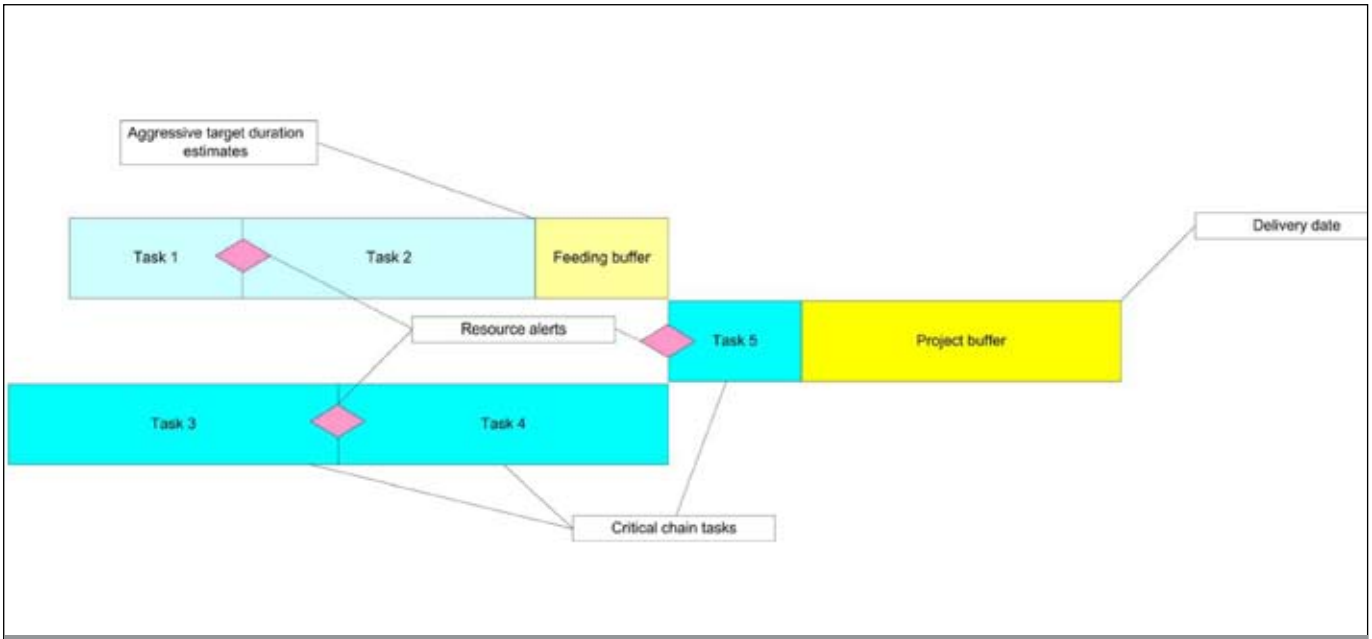


Figure 1 – Feeding buffer and Project buffer

task due dates with 'safety intervals'. Then, by a strange paradox, from the point of view of the whole project, we waste that safety interval due to the apparent comfort it provides, and therefore we put the project delivery day promise in jeopardy.

How to solve these problems? We have to answer the following questions:

• How can we protect the delivery date of the whole testing project from the 'unpleasant surprises' and uncertainty without nailing all the tasks to deadlines on a calendar, which brings Parkinson and wasted safety time into the picture?

• How can we take advantage of early test task finishes when they can help us to accelerate the testing project and maybe allow us to finish it early?

• How can we manage the execution of a testing project, if we don't have due dates to track?

Goldratt's Critical Chain solution

One solution to these challenges is the 'Critical Chain' approach to project management advocated by Eliyahu Goldratt, the father of the Theory of Constraints, in its book Critical Chain [1].

Three things can help us to avoid Parkinson's law:

• Build the schedule with target durations that are enough optimistic to allow/ encourage diversion of attention.

• Get rid of task due dates.

• Charge management with the responsibility to protect project resources from interruptions rather than getting in their way w his now leads directly to and supports the second requirement for repealing Parkinson's law - the elimination of due dates.

In a testing project, there are two kinds of resources: resources that perform critical tasks and resources that perform non-critical tasks. The ones we really have to worry about in this context are the critical chain tasks, since they most directly determine how long the testing project will take. We want to make sure that critical chain resources are available when the preceding task is done, without relying on fixed due dates.
There are two steps required to accomplish this:

1.Ask the resources how much of an advance warning they need to finish up their other work and shift to interruptible work so that when the preceding project task is complete, they can drop what they're doing and pick up their critical task.

2. Require resources to provide regular, periodic updates of their current estimate of the time to complete their current task. When the estimate to complete task T1 matches the advance warning needed by the resource on task T2, let the T2 resource know the work is on its way and that it should get ready to pick it up.

Compared to traditional project management, this is different from focusing on "delivery day" via reporting percent of work complete to focusing on how much time is left to accomplish unfinished tasks.
This process puts us into a position such that we're no longer nailed to the calendar through due-dates, we can move up activity as its predecessors finish early, and we can avoid the impact of Parkinson's law.

But we have not solved completely the first challenge (the part about protecting against 'unpleasant surprises'). We've now got a tight schedule supported by these resource alerts to assure that the critical resources are available when needed and that they can pick up the work when testing tasks are finished earlier than expected.
The problem is that these "50% estimates" don't do too much to help us promise a final due date for the project. We need to protect the due date from variation in the tasks, again, especially critical tasks.
Let's try to shift the safety associated with the critical tasks (fig.1 – tasks 3, 4 & 5, in dark blue) to the end of the chain, protecting the project real due date from variation in the critical chain tasks. This concentrated aggregation of safety is called a "project buffer. (fig.1 – the yellow area placed after the task 5)"

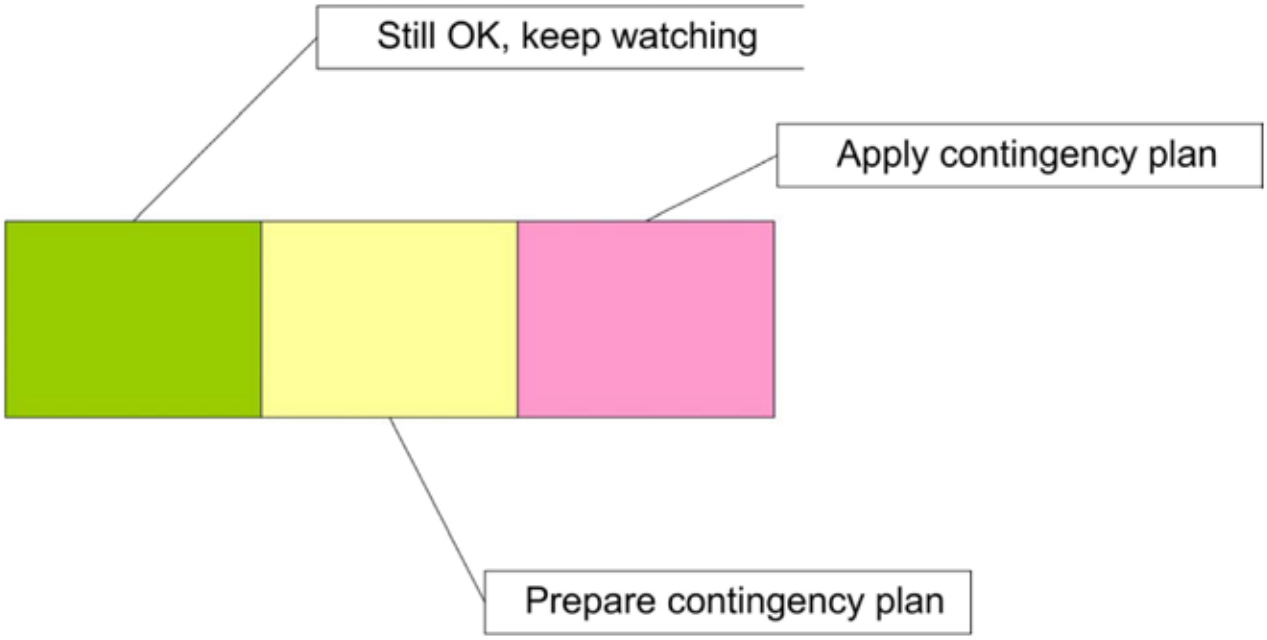


Figure 2 – Buffer management

Now let's turn to the non-critical tasks (fig.1 – tasks 1 & 2, in light blue). Let's assume that they're also allowed to focus on the task at hand and pass it along as soon as it is done - which should be a global model if we really want to get projects done in a timely fashion. But we don't want to micro-manage everybody to the degree we do the critical tasks with the resource availability alerts. Yet we do want to assure that, if things go wrong in the non-critical, we don't want them to risk the ability of the critical tasks to stay on track.

Figure 1 – Feeding buffer and Project buffer

The traditional approach is to start these tasks as early as possible, and hope that the slack or float is enough to absorb the variability. Why not use the buffer approach like we did with the critical chain and the project due date? In this case, concentrate the safety associated with chains of non-critical tasks as a buffer protecting the start of the critical chain task they feed into -"feeding buffers.(fig.1 – the light yellow area placed after the task 2)"

Note that the feeding buffers are also used upon to deal with resource timeliness for non-critical tasks/resources; we don't use the "work-coming alerts" because even if the feeding buffer is consumed, the worst case is that the critical tasks are delayed and maybe eat some project buffer. The feeding, non¬critical tasks are two buffers away from impacting the project promise.

Also, you gain more by keeping non-critical resources focused on the work at hand and to assure they finish work that can be passed on to other resources rather than interrupt them for other non-critical stuff.

Ok, but again, how do we know what shape our test project is in once it gets started, if we don't have due dates to track?
The key is the set of feeding and project buffers and a process known as "buffer management" (fig 2), in 3 steps:

1. As long as there is some predetermined proportion of the buffer remaining, all is well.

2. If task variation consumes a buffer by a certain amount, we raise a flag to determine what we might need to do to if the situation continues to deteriorate.

3. If it deteriorates past another point in the buffer, we put those plans into effect.

Figure 2 – Buffer management

This process allows us to stay out of the way of the test project resources if things are on track, build a contingency plan in something other than a crisis atmosphere, and implement that plan only if necessary.

Conclusions

Goldratt's Goldratt's 'Critical Chain' scheduling model is not a magic 'silver bullet' to guarantee that all the problems will be solved, it's not recommended for all software life-cycle models (for example, an agile project most probably will not benefit from the use of it), but it is a valuable asset in the test manager's best practices collection!

Try to use it only when confronted with a large iterative/incremental software project, having lots of correlated testing tasks, where the 'Critical Chain' scheduling model can be used in testing tasks scheduling in order to guarantee the on-time delivery of the testing activities and to optimize the testing resources usage.

References

1. Eliyahu Goldratt – Critical Chain, North River Press

2. Rex Black – Managing the testing process, Wiley publishing

3. Rex Black – Critical Testing Processes, Addison Wesley

Engineering Productivity: Some ways to measure it and manage it.

advanced

Author: ***Tom Gilb***

About the author:

Tom is the author of nine books, and hundreds of papers on these and related subjects. His latest book 'Competitive Engineering' is a substantial definition of requirements ideas. His ideas on requirements are the acknowledged basis for CMMI level 4 (quantification, as initially developed at IBM from 1980). Tom has guest lectured at universities all over UK, Europe, China, India, USA, Korea – and has been a keynote speaker at dozens of technical conferences internationally.

www.gilb.com, twitter: @imTomGilb

Abstract

There are often too few qualified engineers. I am mostly referring to product design engineers – software engineers and systems engineers. One reason we have too few is that we misuse their time so badly – we waste at least 50% of it. But when we can longer desire or afford to solve the problem by hiring or off-shoring to get more warm-bodies, we need to consider getting more productivity from the engineers we already have. There is one great advantage from that tactic – they already have plenty of experience in our company! There are several tactics to improve productivity. They can take many years to come to full effect, but a steady long term improvement, and dramatic short term improvement, should be possible. The key idea in this paper is that we can define our own productivity quantitatively – and manage the improvement of it quite systematically. Your own definition of productivity demands several simultaneous dimensions of productivity. The definition of productivity also requires substantial tailoring to your organization, and to its current environment. I am going to assert that the best short term measure of engineering productivity is agreed value (requirements) delivered; and the best long term measure of engineering productivity is stakeholder benefits actually delivered.

The Engineering Productivity Principles:

Here are some basic suggestions for a framework for getting control over engineering
productivity:

1. Subjective Productivity: Productivity is someone's subjective opinion of what values we want to create for our critical stakeholders.

2. Measurable Productivity: Productivity can be defined as a setof quantified and measurable variables.

3. Productivity Tools: Productivity can be developed through the individual competence and motivation, the way we organize people, and the tools we give them.

4. Avoid Rework: The initial attack on productivity improvement should be reduction of wasted effort

5. Productive Output: The next level of attack on productivity should be to improve the agreed value delivered to stakeholders.

6. Infinite Improvement: Productivity improvement can always be done: there are no known limits.

7. Perfection Costs Infinity: Increasing system performance towards perfection costs far more than increasing volume of system function.

8. Value Varies: Product attributes are viewed and valued quite differently even by members of the same stakeholder group.

9. Practice Proves Productivity: You cannot be sure how well a productivity improvement strategy will work until you try it in practice.

10. Productivity Dwindles: Yesterday's winning productivity tactic may not continue to work as well forever.

Defining Productivity

Let me tell you what I think productivity is, maybe even what 'engineering' is.

Productivity is delivering promised value to stakeholders.

„Deliver" means actually measurable handed over and available to stakeholders.

„Promised" means that clear written agreements, are made in contacts, requirements, documents and slides, or clear undeniable expectations are set.

„Value" means something of perceived use, to the stakeholder; they need it, they want it, they are willing to sacrifice resources to get it, they will be unhappy if it is late or lower in power than their expectations.

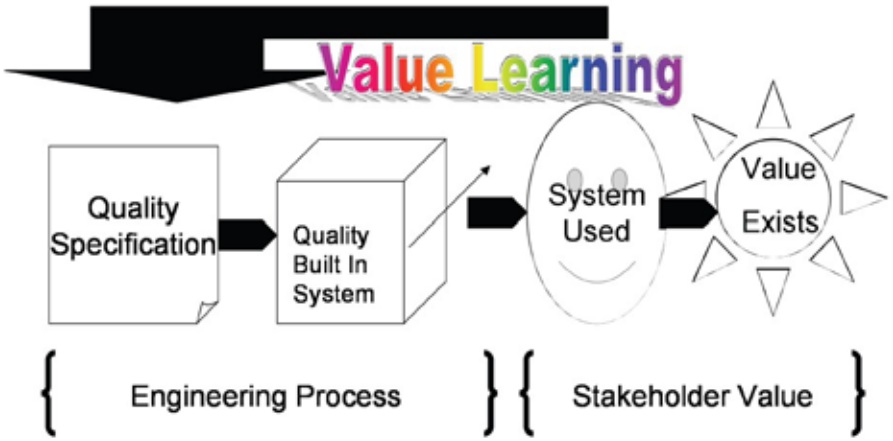


Figure 1

„Benefits" are the results of the perceived value to stakeholders. Benefits are what really happens, though time, as a result of the engineering value delivered.

It is an open question whether systems engineering should attempt to take some planning responsibility for enhancing benefits realization, or whether this is the system recipient stakeholders that should be responsible for planning an environment to maximize benefits.

Someone has to take this responsibility, and I fear that the system users with their 'day jobs', do not feel they are responsible or capable. In which case an opportunity for systems engineers, to enlarge their conventional scope of planning, exists.

So, we can simplify and say 'engineering productivity' is the **ability to deliver agreed requirements.**

Our formal **requirements**, should ideally be the 'meeting place' for stakeholder values and engineering commitments.

An engineer is **productive** to the degree they contribute to an engineering effort that is successful in delivering promised requirements, to real stakeholders, in a timely manner (at or before agreed deadlines).

An engineer is more **efficient** if they can

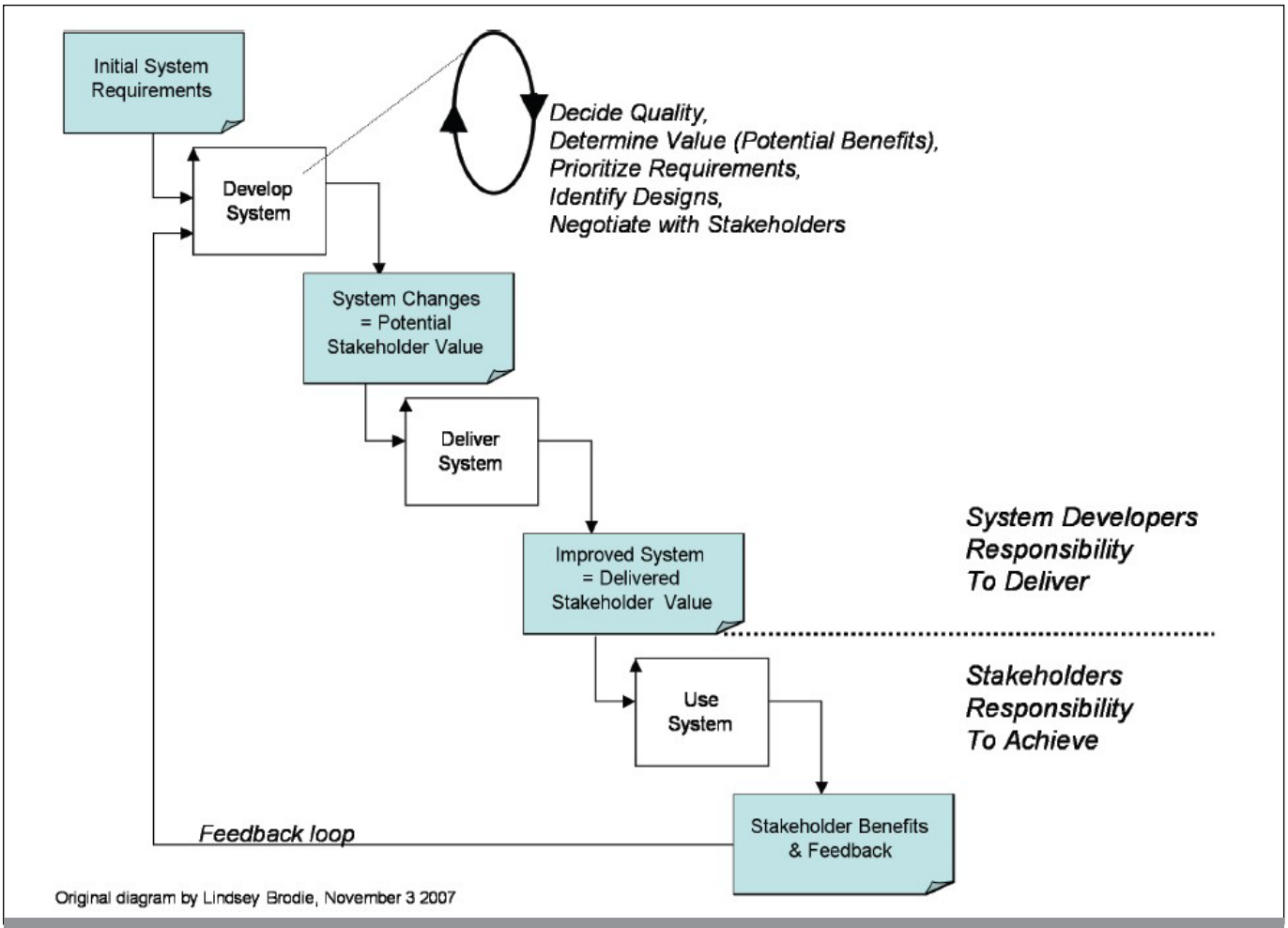


Figure 2

reduce the resources needed to deliver requirements on time to stakeholders.

Stakeholders are any people, groups of people, types of people, or instances that have requirements (like laws, contracts).

Engineers are technical people who, as a team, master the arts of
• determining a necessary set of requirements for a system,
• determining a necessary set of solutions, and
• planning and carrying out the necessary processes to actually delivering the promised requirements (the value, the potential benefits) to the stakeholders.

Figure 1 Engineers can be productive by generating the conditions for stakeholders to get value from the system. The question is, does the systems engineering responsibility stop at the technical system? Or, should it extend into the stakeholder domain? Should systems engineers at least plan (engineer) everything necessary to get the intended value in practice? Is it 'good enough' that value perception exists, but the benefits are not finally brought in, in practice? The next diagram adds a stage, regarding bringing in the benefits.

Figure 2 This diagram makes the subtle distinction between handing over 'potential value' systems to stakeholders, (perhaps this is the limit of engineering responsibility?) and, then actually achieving the full long-term benefits that system deployment enables the stakeholder to do. The rectangle with a left arrow up, is a PDSA process, a Planguage symbol for a process in general.

What Engineering Productivity is not.

1. Not Zero Results: any failure to actually deliver the value agreed, no matter what the reason, or source of cause, means that the engineers have failed to be productive (even if it is not their 'fault').

2. Not Specs: productivity is not the ability to generate specifications of any kind. Specs are perhaps a necessary 'means', but the 'value' delivered is the key notion of real engineering productivity.

3. Not Exceeding Value: productivity is not exceeding agreed requirements, if there is no value, and no agreement with stakeholders.

4. No Golden Hammer: there is no one tool, method, principle or policy that will give you fullpotential productivity: there are masses of details, and persistent improvement, and maintenance forever, that are necessary ingredients.

Some ways to measure engineering productivity

Direct Measures

Value Delivered:
% Lifetime Value Actually Delivered.

This is a summary of all measured or estimated real value delivered to real stakeholders for a defined time period, usually to date. This is % of plans made, of requirement targets that were set.

Potential Value Extrapolation:
% Lifetime Benefits Estimated achievable, under given conditions, based on real measurement and deployment to date.

This is our best estimate of the capability of the system to deliver planned benefits in the longer term, based on real experience of some real stakeholder deployment thus far. The set of future conditions for reaching these estimates, such as budgets, and access to skilled engineers and managers, willingness of stakeholders to continue use, market conditions; need to be spelled out clearly. If prudent, then steps need to be taken. to ensure those conditions are true, as far as we can exercise control over them.

Indirect Measure and Indicator

Technical Capability:
% of Target-Level Improvement of Performance Requirements that is Measurably Delivered

This indicates that the technical engineering work is succeeding. It does not measure that the technical capability has been converted into stakeholder value (deployed at the stakeholder). It could be that the technical system is not yet deployed to stakeholders, except in pilot versions.

Some strategies to increase engineering productivity

Primary Strategies for Value-Delivery Productivity

1. Measuring Value as a strategy
It is all too common, in the many international industries I am personally witness to, that many of the acknowledged critical factors that determine value are not expressed in quantified terms. This seems to be a problem for both management and engineering cultures. We are taught a selection of metrics, for accounting and engineering, but we are not taught that all critical factors must be dealt with quantitatively, even if we have to invent suitable metrics. Senior managers and engineers are not taught, and they do not know how to quantify the very factors they have just acknowledged are critical to the project at hand. They use words, but not numbers.

Examples of real, fuzzy, critical, top level, project objectives

Technical Goals: "rock-solid robustness", "to dramatically scale back the time frequently needed after the last data is acquired to time align, depth correct, splice, merge, recompute and/or do whatever else is needed to generate the desired products by semi-automating and/or performing these activities as the data comes in", "to make the software much easier to understand and use than has been the case for previous software", "to provide a much more productive software development environment than was previously the case.", "software development difficulty should scale", "will provide a richer equipment model that better fits modern hardware configurations", "Minimal down-time", "major improvements in data quality over current practices wherein the job planning process is much more haphazard."

Business Systems: "Business Result Alignment: maximize delivery speed and client satisfaction level across the Change the Firm Book of Work to achieve key business goals.", "Eliminate IT efforts that duplicate other IT efforts.", "Make use of existing tools and avoid reinventing the wheel", "Deliver high-significance real-time metrics on critical aspects of project results and resources.", "to be the world's premier integrated service provider" (in our sector).", "a much more efficient user experience"

Engineering Organization Objectives:

A special effort is underway to improve the timeliness of Engineering Drawings. An additional special effort is needed to significantly improve drawing quality. This Board establishes an Engineering Quality Work Group (EQWG) to lead Engineering to a breakthrough level of quality for the future. To be competitive, our company must greatly improve productivity. Engineering should make major contributions to the improvement. The simplest is to reduce drawing errors, which result in the AIR (After Initial Release) change traffic that slows down the efficiency of the manufacturing and procurement process. Bigger challenges are to help make CAD/CAM a universal way of doing business within the company, effective use of group classification technology, and teamwork with Manufacturing and suppliers to develop and implement truly innovative design concepts that lead to quality products at lower cost. The EQWG is expected to develop 'end state' concepts and implementation plans for changes of organization, operation, procedures, standards and design concepts to guide our future growth. The target of the EQWG is breakthrough in performance, not just 'work harder'.
The group will phase their conceptualizing and recommendations to be effective in the long term and to influence the large number of drawings now being produced by Group 1 and Group 2 design teams.

Example 1 Real example from a 5,000-engineer corporation (1989). Source: CE, page 71, Case 2.8 where a detailed analysis of this text is given. In this case the Director for Productivity and Quality for Engineering was denied about $60 million from the Board, to fund this project (which was to buy more automation of engineering work processes). He was quite surprised, because in the past, this level of proposal had worked! Can you work out the proposed value of the investment from this?

The quoted examples are real (1989-1998-2006-2007 vintage), and reflect real projects where the $50 million in one case, and $100 million (in another case) ***actually spent*** was totally ***wasted***, no value delivered at all. In the last example, the Board was smart enough to NOT waste the money!

The major initial culprit, in my opinion, was lack of quantification of these management-acknowledged, top-level, large project, objectives. At least one top manager in each case totally agreed with my conclusion. The root cause of this bad practice, in my opinion, was lack of corporate policy, regarding quantification of top-level objectives for big projects. There was no common-sense culture (to make up for the lack of formal culture), amongst the managers approving the 'investment', to acknowledge that the objectives were on very shaky ground.

2. Estimating Long Term Value – strategy
We are all familiar with the 'business case'. A typical business case will probably insist that we feed it with some monetary figure regarding long-term savings, or additional earnings as a result of the investment in the project (monetary value) – the 'benefits'.

The problem with this, is it is not ever based on a detailed analysis of the many stakeholders, and their value set. It might even typically ignore all stakeholders except 'us' ourselves. It will probably focus entirely on monetary advantages, and seriously ignore all other advantages, even though the other advantages may well be listed as 'Critical Business Objectives' (see above examples, strategy 1).

In addition, there may be no obligation, culture, will-power, or ability to actually follow-up and derive the projected benefits in practice. Last month I was told frankly at one place I visited, that although projects said in project justifications, for example, they would "save 20 employees", they were routinely never actually saved, and everyone knew there were no penalties for failing to make the saving real, when new systems were delivered.

A respectable strategy would be to make estimations of long-term benefits expected for all aspects of value, for all stakeholders of significance. We should of course include information on the conditions and assumptions necessary for these benefits to be realized in practice.

3. Focus on Delivery of Value to Stakeholders – strategy
We have a tendency to focus on value to our corporation; the one investing in the project. Or we focus on value to our main customer, paying for the project.

We have to shift culture, to a time-honored systems engineering notion, that of the many project stakeholders [SEH, references in 80 sections to stakeholder]. Each one of say 40 stakeholders will have one, or probably more, value delivery potentials from the project. We need to map all significant stakeholder values, even though they are not 'ours'.

These values are not the same at requirements! Stakeholder values represent potential requirements if they are technically possible, economically possible and prioritized! They are, for the moment, just stakeholder needs and values, not committed system requirements.

The engineers doing this will increase their real 'productivity' by helping to plan the actual delivery of those values. And perhaps even contribute to planning the total systems problem of delivering real benefits on the back of the values deployed technically.

We need to plan to help stakeholders and inform stakeholders, and get co-operation of many of those stakeholders, so that they understand and commit to their role in deriving those final benefits for themselves, and for other stakeholders.

Example 2 a design template, partly filled out in Planguage (Real, telecoms, about 2000). It has collected information on defined stakeholders that are impacted by this design. It has identified a critical technical requirement (Interoperability) impacted by this design. It has identified a critical technical requirement (Interoperability) impacted by this design. It has yet-unfilled parameters about impact relationships, that challenge us to enrich our understanding of this engineering artifact. The engineer can increase their productivity by analyzing deeper, and acting on the analytical insights. It is not about producing more, but about producing more potentially-fruitful insights for engineering and managing value to stakeholders. Source [CE], page 199.

Secondary Strategies: that will improve our ability to deliver value.

Quantifying Performance, particularly qualities.
Technical system qualities, are not the same is the stakeholder value we discussed above. The technical qualities are the pre-requisites, or 'drivers', of value. But qualities are not the value derived finally by stakeholders.

Example of a Design Specification

Tag: OPP Integration.
Type: Design Idea [Architectural].
========== Basic Information ==========
Version:
Status:
Quality Level:
Owner:
Expert:
Authority:
Source: System Specification Volume 1 Version 1.1, SIG, February 4 – Precise reference <to be supplied by Andy>.

Gist: The X-999 would integrate both 'Push Server' and 'Push Client' roles of the Object Push Profile (OPP).
Description: Defined X-999 software acts in accordance with the <specification> defined for both the Push Server and Push Client roles of the Object Push Profile (OPP).
Only when official certification is actually and correctly granted; has the {developer or supplier or any real integrator, whoever it really is doing the integration} completed their task correctly.
This includes correct proven interface to any other related modules specified in the specification.

Stakeholders: Phonebook, Scheduler, Testers, <Product Architect>, Product Planner, Software Engineers, User Interface Designer, Project Team Leader, Company engineers, Developers from other Company product departments which we interface with, the supplier of the TTT, CC. "Other than Owner and Expert. The people we are writing this particular requirement for."

========== Design Relationships ==========
Reuse of Other Design:
Reuse of This Design:
Design Constraints:
Sub-Designs:

========== Impacts Relationships ==========
Impacts [Functions]:
Impacts [Intended]: Interoperability.
Impacts [Side Effects]:
Impacts [Costs]:
Impacts [Other Designs]:
Interoperability: Defined As: Certified that this device can exchange information with any other device produced by this project.

========== Impact Estimation/Feedback ==========
Tag: Interoperability.
Scale:
Percentage Impact [Interoperability, Estimate]: <100% of Interoperability objective with other devices that support OPP on time is estimated to be the result>.
========== Priority and Risk Management ==========
Rationale:
Value:
Assumptions: There are some performance requirements within our certification process regarding probability of connection and transmission etc. that we do not remember <-TG.
Dependencies:
Risks:
We do not 'understand' fully (because we don't have information to hand here) our certification requirements, so we risk that our design will fail certification <-TG.
Priority:
Issues:
========== Implementation Control ==========
Not yet filled in.
========== Location of Specification ==========
Location of Master Specification: <Give the intranet web location of this master specification>.

For example if a system is designed to have a security quality of identifying 99% of attempted system intrusions within 1.o seconds, a 'quality level [Security Quantification]. There is no value if the system is not yet deployed, and if it has no effect on the hacker activity (because no hackers are aware of the capability, and choose to avoid the system), or if no hackers are caught in the act.

For another example, if a system is designed for high usability, in order to make it unnecessary to train people for a week on the use of the system, but an organization persists in delivering the useless training in spite of this, then no value is actually delivered to the stakeholder. The potential is there, but not exploited.

Now, just as the above (1. Measuring Value as a strategy), argues that we cannot expect to engineer the value achievement, if the value aspects are not defined quantitatively, the same argument applies, for the same reasons, at the level below stakeholder values, the system quality levels.

System quality levels must be quantified by engineers, and must be engineered into existence. That is a minimum prerequisite for enabling the system to deliver value to stakeholders. [QQ].

Figure 3 the engineering-specification structure of a single quality-aspect (Repair) of a system. This quality aspect would have no value to any stakeholder if the system was never deployed or released, or never had a fault needing repair, or if repair activity were never attempted, or if it were not attempted using the technology designed in the system to give this repair speed. Technical qualities are the basis for deriving value, but they are not to be confused with the value ('perceived potential benefit') itself, or even with the long-term benefits ('value delivered to stakeholders') derived from the quality of the system. Source: [CE, SoM] Figure 4.3, page 115.

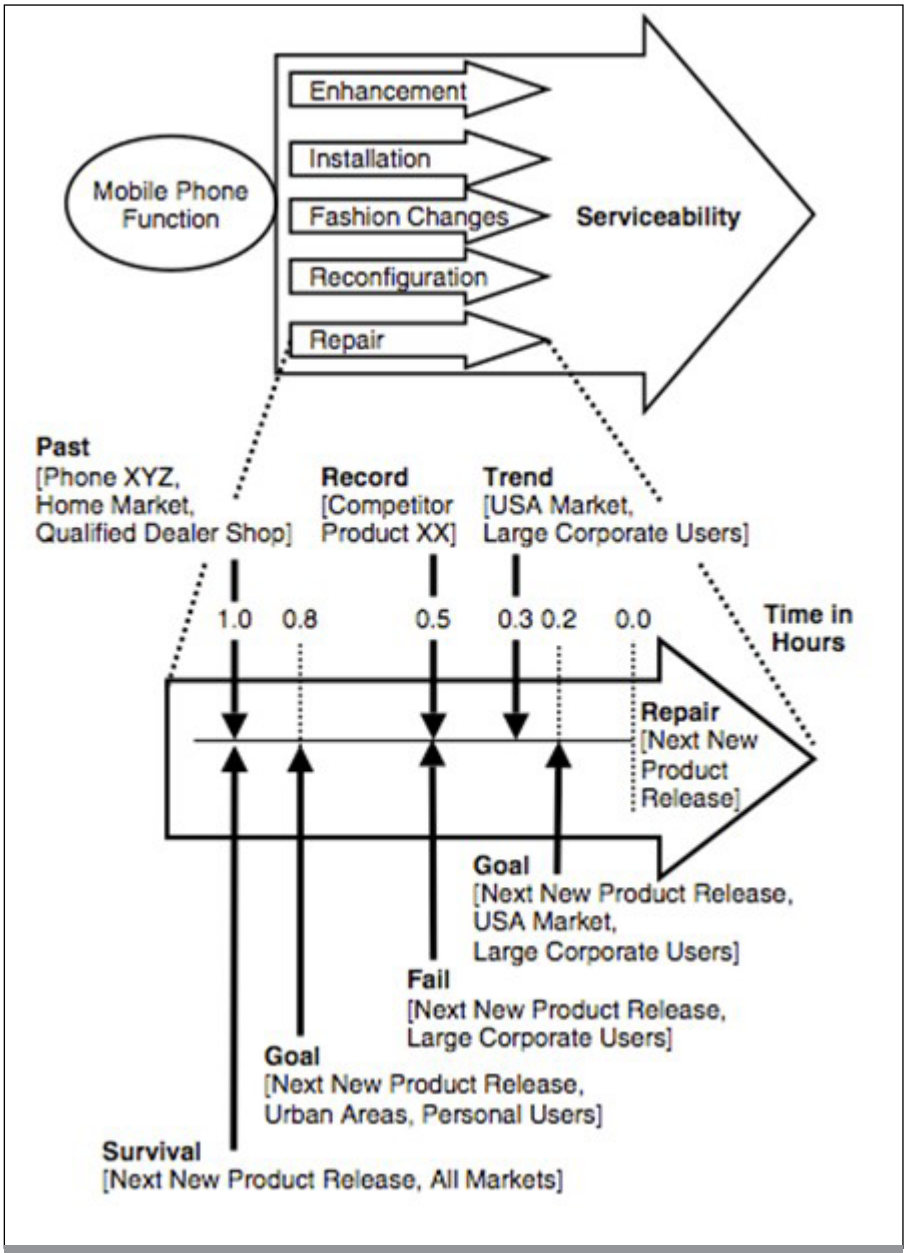


Figure 3

Evolutionary Project Management, feedback and correction.

In complex, state of the art, multi-stakeholder, large-scale systems it is acknowledged [US DoD Mil Std 498, for example] that it is impossible to know all the right requirements at the beginning. We have to learn more about, and adjust, initial assumptions, as realities emerge.

From my perspective a major tool to help the systems engineer dialogue with the reality of both the technical, political, economic and other stakeholder environments, is that we create an engineering process that learns. The engineering process learns about stakeholder values, about necessary and possible requirements, about emerging technology, about the real ability to make benefits happen, and many other uncertain variables. The engineering process learns early, frequently, and is narrowly focused – not distracted by overwhelming size and complexity.

The class of project management methods that do this are broadly known as 'evolutionary' methods. These are iterative, they are incremental; but they have one more attribute that makes them 'evolutionary': feedback on each cycle, learning, and corrective action to benefit from the feedback and analysis. In short they are also 'learning' processes.

Although it is not difficult to see this kind of gradual learning process, in many forms, in engineering (multiple prototypes, multiple product versions, the long term evolution of most technologies), current systems engineering culture does not take such processes for granted at all. If anything, we have got a systems engineering culture that largely assumes something closer to a 'waterfall' model of development [SEH]. It hardly mentions evolutionary processes at all.

I would argue that a systems engineer must normally use, and master an

Figure 4

evolutionary feedback project mechanism [Evo]. The fact that corporations and institutions routinely impose a heavy bureaucratic 'big bang' model, with attendant project failures, is a sorry comment on our present culture.

Figure 4 A process-improvement cycle: "Understand-Select-Analyze-Plan-Do-Check-Act" which emphasizes that the plan must be based on the understanding of the system and the evaluation of the data on the system. We need to apply these cycles better to project management. Source: http://www.triz-journal.com/archives/1998/12/g/Image99.gif

One of the main conclusions Peter Morris made, in his great book on project management [Morris] was that there was "no good project management method". He was talking about projects like the Concorde, The Channel Tunnel, and the Atomic Bomb (Manhattan). He was talking about systems engineering. His main conclusion was that if we are to improve the project management model, it must include much more feedback – an evolutionary model. Systems engineering has not yet taken his advice to heart. Our SE culture is too slow to react to necessities.

One of my favorite tools

Impact Estimation Tables

I believe that the productive engineer needs another tool, which I have called the Impact Estimation table [CE], or a similar tool such as Quality Function Deployment (if it is carried out with the same quantified rigor in specification – rare to see in [QFD] practice – but I am told it exists). We need to be able to reason about complex systems, and about the value we are planning to deliver

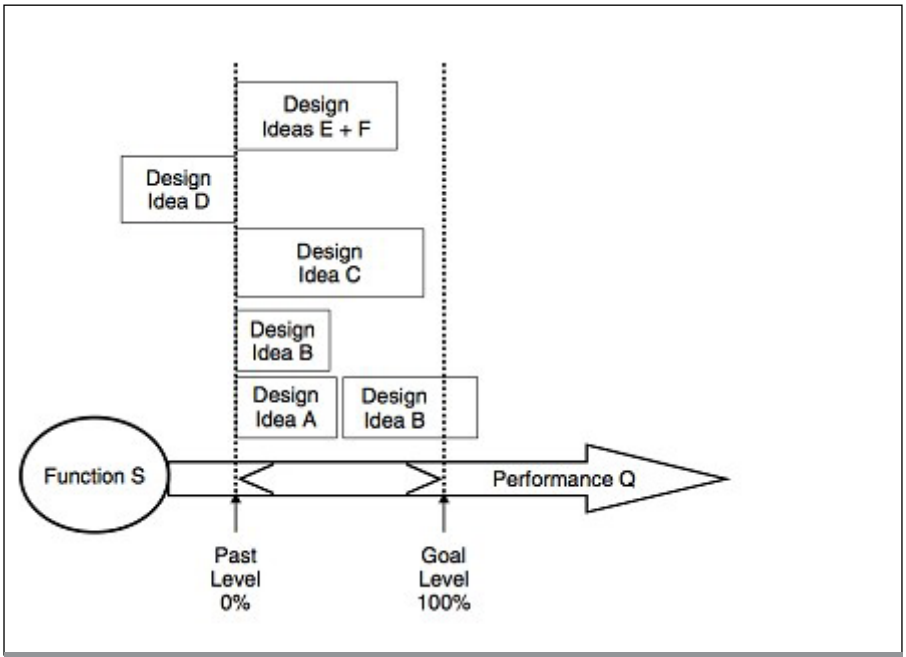


Figure 5

as a result of our technical engineering.

Figure 5 The connection between design (for example, required technical system qualities) and Performance Goals (for example derived stakeholder value levels) can be both estimated, and later measured. The estimated or achieved value can be represented graphically, as above (in 'Planguage, [CE]) or on spreadsheet tables.

We need to avoid the common one-to-one reasoning ('we are going to use technology X to achieve Quality Y') and to understand more clearly that our means are likely to have multiple effects on many of our critical values. This is, of course, good conventional engineering (to worry about side effects) but I see too many real projects where this is not done systematically.

My opinion is that the use of a tool like the Impact Estimation table, would force the systems engineering team to consider their systems, as broadly as we must do in a real systems engineering environment.

Figure 6 A real US DoD Impact Estimation table, from the author's client, the Persinscom (US Army, Personnel System). Behind all tags (Customer Service, Technology Investment) are properly-defined requirements (quantified) and designs. This tool, enables us to get a better overview picture of how mutiple technological ideas, Source CE, page 284.

Some Management Policies for Engineering Productivity

1. Productivity is Value Delivered: SE Productivity is ultimately measured in terms of real benefits delivered to real stakeholders, as enabled by stakeholder value delivered, which is the short term

Design Ideas ->	*Technology Investment*	*Business Practices*	*People*	*Empowerment*	*Principles of IMA Management*	*Business Process Re-engineering*	*Sum Requirements*
Customer Service ? <->0 Violation of agreement	50%	10%	5%	5%	5%	60%	185%
Availability 90% <-> 99.5% Up time	50%	5%	5–10%	0%	0%	200%	265%
Usability 200 <-> 60 Requests by Users	50%	5–10%	5–10%	50%	0%	10%	130%
Responsiveness 70% <-> ECP's on time	50%	10%	90%	25%	5%	50%	180%
Productivity 3:1 Return on Investment	45% 50%	60% 5%	10% 75%	35% 45%	100% 15%	53% 61%	303% 251%
Morale 72 <-> 60 per month on Sick Leave							
Data Integrity 88% <-> 97% Data Error %	42%	10%	25%	5%	70%	25%	177%
Technology Adaptability 75% Adapt Technology	5%	30%	5%	60%	0%	60%	160%
Requirement Adaptability ? <-> 2.6% Adapt to Change	80%	20%	60%	75%	20%	5%	260%
Resource Adaptability 2.1M <-> ? Resource Change	10%	80%	5%	50%	50%	75%	270%
Cost Reduction FADS <-> 30% Total Funding	50%	40%	10%	40%	50%	50%	240%
Sum of Performance	*482%*	*280%*	*305%*	*390%*	*315%*	*649%*	
Money % of total budget	15%	4%	3%	4%	6%	4%	36%
Time % total work months/year	15%	15%	20%	10%	20%	18%	98%
Sum of Costs	*30*	*19*	*23*	*14*	*26*	*22*	
Performance to Cost Ratio	*16:1*	*14:7*	*13:3*	*27:9*	*12:1*	*29:5*	

measure of engineering productivity.

2. Total Systems Engineering: The engineering organization is responsible for all aspects of value delivery; if necessary including the design of the organization needed to continue to deliver the real benefits in the long term.

3. Value Responsibility: specified engineering organizational units will be held accountable for initial and long term planned value delivery.

4. CVO: A Chief Value Officer will oversee all technical and management efforts on value delivery; and report to the CEO on the situation, using Value Accounting.

Summary

We need to develop a culture in systems engineering, where the delivered value and consequent benefits are considered the primary purposes of systems engineering. Value to stakeholders can be a primary measure, short term, of the productivity of systems engineering. „Delivered benefits" is a better measure of the real productivity of the systems engineering function.

References

[CE] Gilb, Tom, Competitive Engineering, A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage, ISBN 0750665076, 2005, Publisher: Elsevier Butterworth-Heinemann. Sample chapters will be found at Gilb.com.
as noted below:

[SoM] Chapter 5: Scales of Measure:

http://www.gilb.com/community/tiki-download_file.php?fileId=26

[Evo] Chapter 10: Evolutionary Project Management: http://www.gilb.com/community/tiki-download_file.php?fileId=77

Gilb.com[www]:www.gilb.com.ourwebsite has a large number of free supporting papers , slides, book manuscripts, case studies and other artifacts which would help the reader go into more depth.

[QQ] Quantifying Quality theme:
[QSV] Quantifying Stakeholder Values (INCOSE 2006 paper) http://www.gilb.com/community/tiki-download_file.php?fileId=36
[H2QQ] Main QQ paper 2007 Version
"How to Quantify Quality: Finding Scales of Measure". http://www.gilb.com/community/tiki-download_file.php?fileId=124. This was originally published as a paper at INCOSE.org conference Washington DC 2003.

[QQ Slides] http://www.gilb.com/community/tiki-download_file.php?fileId=131 This is a 2007 version of the slides used to lecture on quantifying quality, at universities and conferences, and for clients.

[QFD] What's Wrong with Quality Function Deployement?

http://www.gilb.com/community/tiki-download_file.php?fileId=119

[Security Quantification] Quantifying Security: How to specify security requirements in a quantified way. Unpublished paper. Tom Gilb. See also [SoM] above. http://www.gilb.com/community/tiki-download_file.php?fileId=40

[Morris] The Management of Projects, Peter Morris, UMIST, ISBN: 9780727716934, 1994, Thomas Telford Ltd., 358 pp, http://www.thomastelford.com/books/bookshop_main.asp?ISBN=072771693X

[SEH] INCOSE Systems Engineering Handbook v. 3.
INCOSE-TP-2003-002-03, June 2006 , www.INCOSE.org ■

Test Process Maturity and Related Measurement

intermediate

Author: ***Nagaraj M Chandrashekhara***

About the author:

Nagaraj is Director-Customer Excellence, in STAG Software Private Ltd. He has over twenty-five years of experience in the software discipline and has worked in all the phases of software development life cycle out of which sixteen years in the field of Software Test Engineering.

He is passionate about driving process improvement for achieving better business results. He has delivered lectures on Software Engineering, Software Testing and Software Process Improvement at many organizations and in International Testing Conferences. He was certified internal auditor for ISO and part of internal assessment team for CMM model.

He holds an engineering degree in Mechanical discipline from Mysore University. He has completed diploma in Statistical Quality Control (SQC) from ISI. His interests are in the areas of test design, project management, test process improvement using TMM, and data analysis using statistical tools. He had submitted papers and tutorials on different topics of his interest in the international conferences held in India, Singapore and Malaysia.

Contact: nagaraj@stagsoftware.com

Abstract

The organizations have moved from "assemble testing team on need basis" to "a focused independent testing team" over years. It is quite natural that the test process to run the organization also has changed drastically to address the entire test life cycle process covering different aspects of process engineering.

Organization involved in software development normally chooses standard models as framework to improve their software development life cycle process like ISO 9001, CMMI, SPICE etc. Because of the important role of testing in software process and product quality, and the limitations of existing process assessment models, Ilene Burnstein at Illinois Institute of Technology developed the Testing Maturity Model. This model helps organizations to introduce best practices in progressive way and assess the capability and maturity of test process against a set of standards goals.

The quality of measurements collected regarding testing activities improves over the years. This also reflects the level of process maturity achieved. This paper is intended to share authors experience on journey of improvements regarding test related measurement collections observed in the organization over time.

At each level of process maturity the goals are different. The measurements we collect to understand the status of maturity level goals achieved is explained in this paper using GQM model.

Introduction

We all know, the organizations have moved from "assemble testing team on need basis" to "a focused independent testing team" over years. It is quite natural that the test process to run the organization also has changed drastically to address the entire test life cycle process covering different aspects of process engineering. The quality of measurements collected regarding testing activities improved over the years. This also reflects the level of process maturity achieved. This paper is intended to share authors experience on journey of improvements regarding test related measurement collections observed in the organization over time.

The Test Maturity Model is used as framework to explain the growth of maturity in measurement collection and analysis in an organization using TMM as model for process improvement. The paper just introduces on TMM model and maturity level goals in first section. The authors experience in process improvement and how meaningful measurements are recommended to be collected are explained using Goal Question Metrics as model in this paper in the remaining sections.

Test Maturity Model

Test maturity model (TMM) developed by a research group headed by Ilene Burnstein at the Illinois Institute of Technology. The TMM used by many software development organizations to assess and improve their testing process. This model that illustrates in stages how a testing process grow incrementally.

The internal structure of TMM maturity level explained in the picture below.

Fig 1: The internal structure of TMM maturity levels
Courtesy: Practical Software Testing by Ilene Bursnstein

Figure 1

What is good about this framework is identification of three critical players who play a major role in test process improvement. They all have to work together towards the evolution of a quality testing process. These groups were managers, developers/testers, and users/clients. In TMM terminology they are called three critical views. Each groups view the testing process from a different perspective that is related to their particular goals, needs and requirements. The manager's view involves commitment and support for those activities and tasks related to improving testing process quality. The developer/tester's view encompasses the technical activities and tasks that when applied, constitute best testing practices. The user/client view is defined as a cooperating or supporting view. The developers/testers work with client/user groups on quality related activities and tasks that concern user oriented needs. The focus is on soliciting client/user support, consensus, and participation in activities such as requirement analysis, usability testing, and acceptance test planning. At each TMM level the three groups play specific roles in support of the maturity goals at the level.

TMM Levels, Goals and Characteristics

The various levels and related goals are summarized in a tabular form below with observed characteristics of organization at each level.

Goals	Characteristics
Level 2: 2.1 Develop Testing and Debugging Goals and Policies 2.2 Initiate a Test Planning Process 2.3 Institutionalize basic testing techniques and methods	There is a clear separation between debugging and testing phase. It is a planned activity in project plan. Plan starts after coding is complete. Basic testing techniques in place. Testing is multi-leveled.
Level 3: 3.1 Establish a Test Organization 3.2 Establish a technical training program 3.3 Integrate testing into the software life cycle 3.4 Control and monitor the testing process	There is an established test organization. Testing is integrated to SDLC. Test plan is developed, tracked and controlled (Integrated with project plan). Test engineers drive test process improvement. Users/clients attend milestone meeting. User/clients support in developing usability test plans.

Goals	Characteristics
Level 4: 4.1 Establish an organizationwide review program 4.2 Establish a test measurement program 4.3 Software quality evaluation	Review program is effective Reviews are planned activity in project plan. Measurements collection and analysis process effective. Quality attributes of a product are well defined and measured.
Level 5: 5.1 Defect prevention 5.2 Quality control 5.3 Test process optimization	Data from all projects are collectively analyzed. Critical defects types are analyzed thoroughly. Quality control concepts are adopted. Right tools are inserted progressively.

Goal, Question and Metrics

Before we understand the measurements collected at each level of test process maturity in the organization, let us understand the GQM model to arrive at the measurements for all levels. This model recommends that you have to be a goal focused to collect measurement, understand goals and related questions to check the status of goals. The measurement you collect must help to derive metrics to answer possible questions to check on goal status. Just to illustrate this model let us take an example of a typical goal in any project.

Let us assume one of the goals of a project as “deliver project on time”. Let us also assume that the project has 3 major milestones before final delivery to customer treated as 4th milestone. The obvious questions to check the status of the above goal is are we on track? . Normally the measurements we collect will be

- Planned date of milestone completion
- Actual date of completion
- Total planned effort for milestone
- Effort spent for milestone

The above when we collect is just called as data. If we compute number of days late or extra effort spent it is a measurement. If we compute % schedule variance and % effort variance, which represent the attribute of degree of late or

early completion of milestone, are called metrics. Compare these metrics from one milestone to another than we know the trend and risk of meeting final end date. Some of the possible inferences we can make from such metrics is “the delay is consistently above x% and it is tough to meet the end date and unless we take significant corrective actions”.

Measurements seen at different level of test process maturity

The next set of sub-sections explains the author recommended measurements at different TMM levels of test process maturity. A sample questions are explained to get advantage of these measurements and derive a value which explains where we stand with respect to goals set for each maturity level by the model.

Measurement of TMM Level 1:

There is no specific goal here in TMM model. The characteristics of organization where test process maturity is at this level will be:

- Testing is a chaotic process
- Not distinguished from debugging
- Documented set of specification not available
- Tests are developed in ad-hoc way after coding is completed
- The objective of testing is to show that software works

The measurement recommended for TMM level 1 are:

Size:

- Size of code in KLOC
- Number of requirements or features
- Number of test cases developed
- Number of test cases executed

Defects:

- High, medium, low severity defects count
- Defects/KLOC

Cost:

- Costs of the project as whole
- Cost of the testing effort

Measurements for TMM Level 2

As already mentioned in section 3 the various goals at this maturity level are:

- Develop Testing and Debugging Goals and Policies
- Initiate a Test Planning Process
- Institutionalize basic testing techniques and methods

The measurements recommended for TMM level 2 are:

Time/effort related measurement:

- Time/effort spent in test planning
- Time/effort spent in unit, integration, system, regression testing
- Total time/effort spent in testing activities
- (Granularity in above measurements also possible like time/effort spent in test design for unit / integration/ system tests)
- Number of planned test cases, unplanned test cases
- Planned/actual degree of statement coverage

Defects:

- Number of defects in each phase (SDLC Phases)
- Number of defects found in each level of testing (UT, IT, ST, UAT)
- Number of each type of defects found
- Time taken to fix and re-test each defects type

Some sample recommended questions to check on goal status for level 2:

- Do you see % of effort spent on testing and debugging across all projects in organization is improving when compared to last year?
- What percentage of defects was logged at each phases of software development across projects?
- Is % of test planning effort spent in overall effort of a project improving across projects in the organization?
- What percentage of planned measurements is collected in each project?
- Does percentage of engineers trained formally on basic techniques and tools required for effective testing improving every quarter in an organization?

Measurements for TMM Level 3

As already mentioned in section 3 the various goals at this maturity level are:

- Establish a Test Organization
- Establish a technical training program
- Integrate testing into the software life cycle
- Control and monitor the testing process

The measurements recommended for TMM level 3 are:

Coverage related measurement:

Requirement coverage, Statement, branch coverage

Productivity related measurement:

- Number of Test cases written / Per unit time
- Test cases executed /Per unit time

Training related measurement:

Number of training hrs attended / year (for all test professionals)

ROI on tools initiatives:

Total cost saved by automation/ Total tool program cost

Defect escapes at each level of testing:

- Unit test escape = (Total UT bugs found in ST/ Total ST bugs)
- System test escape = (Total bugs found in UAT / Total ST bugs)

User/clients support:

Number of Users/Clients interactions

Some sample recommended questions to check on goal status for level 2:

- What are the percentage people in different levels of test team?
- Is the above percentage growing overtime?
- What is the percentage of people trained on different topic of test engineering across organization? (Look at various disciplines in STEM™ [1]test technology of STAG. Visit: www.stagsoftware.com to know more about STEM™)

[1]STEM™ - STAG Test Engineering Method is a test technology of STAG Software Private Limited

- What percentage of defects was logged at each phases of software development across projects?
- What percentage of projects in the organization test engineers were involved from day one of the project?

• Can you see % of defects uncovered at each stage of SDLC across projects?
• Does the management get visibility to the % of test effort planed VS actual spent, productivity of test engineers, coverage of test cases, coverage of requirement, code coverage, types of tests planned and covered at each review?
• What percentage of project released with acceptable variance to schedule?
• Do we understand the ROI of tool program initiated in the organization?
• What % of defects escaping each quality gate? Author recommends going through more about software cleanliness criteria as explained in STEM™)
• Do we understand user interaction effort as % spent against planned and the impact of users support when really required on quality attributes definition?

Measurements for TMM Level 4

As already mentioned in section 3 the various goals at this maturity level are:
• Establish an organizationwide review program
• Establish a test measurement program
• Software quality evaluation

The measurements recommended for TMM level 4 are:

• Number of inspection leaders available
• Number of people trained on inspection
• Size of the item inspected
• Time spent on inspection activities
• Number of defects found during inspection
• Effort spent on measurement analysis
• Effort spent on different types of tests in a project (quality attributes focus)

Some sample recommended questions to check on goal status for level 4:

• Do we understand various quality attributes defined for cleanliness of software and What % of this we met before release?
• What % of test effort spent on different types of tests planned?
• What % of projects adhered to measurement program fully in the organization?
• What % of defects in project was uncovered by review/inspections?
• What % of people in team trained on formal inspection technique?
• How effective is inspection process? (Defects found per hour of inspection effort)

Measurements for TMM Level 5

As already mentioned in section 3 the various goals at this maturity level are:
• Defect prevention
• Quality control
• Test process optimization

The measurements recommended for TMM level 4 are:

• Time/Effort spent in defect causal analysis
• Number of actions suggested
• Effort/cost for implementing action plans
• Costs of statistical testing
• Effort/costs of training SEPG team in process control
• Effort/costs on quantitative process analysis
• Number of process changes
• Number of new tools introduced in the organization

Some sample recommended questions to check on goal status for level 5:

• Do you see defects counts are classified under few list of types and % spread in a project across these types are captured?
• Do we analyze root cause for critical defects types, which impacted project?
• What percentage of people is trained on process control techniques in the organization?
• How many project attributes are measured and declared as success based on organization control charts?
• What percentage of penetration happened in all tools introduced in the organization?

Conclusions

Test process in an organization will mature overtime. If TMM model is used as framework for test process improvement, the recommended measurements in this paper definitely will help to measure the progress of test process maturity in the organization. Measurement should have a clear goal so that you get the support from all concerned in the organization on collections, analysis and actions to improve test process continuously.

References:

1. Practical Software Testing By Illene Burnstein, Springer
2. Software Metrics: By C. Ravindranath Pandian ■

Comparison of Change Management Systems: ClearQuest, VSTS, Redmine and BugTracker.NET

basic

Author: ***Stanislav Ogryzkov***

About the author:

30 years old. Specialist in enterprise-wide information systems, business process re-engineering (BPR), quality management (including testing). ISTQB Certifed Tester, Foundation Level, and a certified internal auditor of quality management systems (ISO 9000).
Graduate of Vladimir State University, Russia. MSc major in computer science, PhD major in technical science. One of the two first ISTQB Certifed Tester in Russia.
Since 2004: Quality Assurance Person at Inreco LAN inrecolan.com), an offshore software development outsourcing company located in Vladimir, Russia. Since 2005: Quality Assurance Manager at the same company. Since 2006: Business Process Improvement Manager at the company. At last, since 2010, Chief Information Officer (CIO) at Inreco LAN.
See http://stanislaw.ru/eng/author/resume.asp for details.

Today software development industry is impossible without using change management systems which help to register all changes in the software developed or supported, to plan and track them. Here "changes" means different changes in graphical design, software architecture, source code, user experience, integration features, etc., particularly:

• **defects**, or **bugs** – detected unconformity of the software towards the requirements that must be fixed;

• **enhancements**, or **features** – new features, properties, etc. of the software, improving its functionality, quality and/or usability;

• **tasks** – various tasks in the software development or support project, for example, a task of configuring source code backup.

I do not plan to discuss the need of using such systems in software development practices as I consider it an obvious fact (moreover, they are useful in other industries). I am going to tell you about four such systems, listed in the order of decreasing "weight":

• Rational ClearQuest is a part of a mega package called Rational Suite and is a "native" tool to apply the Rational Unified Process (RUP) methodology; we studied it in our university within software development disciplines and later evaluated it in some software development projects for American customers.

System	Price	Language	Web Interface	Lifecycle	Authentication	E-mail	Reports	Source Control Integration	User-Defined Fields	Database Server
ClearQuest	$1810	En	Yes	Configurable	Native	Excellent	CSV, Excel	Native source control (ClearCase)	Yes	DB2, Oracle, SQL Server, Sybase
VSTS	$13790	En	Yes, TeamPlain	Configurable	WAD	Satisfactory, excellent when used with TeamAlerts	Excel, HTML, Project, Reporting Services	Native source control, Subversion integration is possible	Yes	SQL Server
Redmine	$0	En, Ru, etc.	Yes	Configurable	Native/WAD	Satisfactory	Atom, CSV, HTML, PDF	Bazaar, CVS, Darcs, Git, Mercurial, Subversion	Yes, without operations over them	MySQL, PostgreSQL, SQLite
BugTracker.NET	$0	En	Yes	Complete graph	WAD/native	Good	Excel, HTML	Subversion	Yes, 3 drop-down lists	SQL Server

Figure 1

• Microsoft Visual Studio Team System (VSTS) 2008 Team Suite, also known by the name of its part, Team System Foundation Server (TFS), became available to us as an independent software vendor (ISV) who reached the status of Microsoft Gold Certified Partner.

• Redmine is one of free alternatives, closer to systems with a wider functionality – project management systems.

• BugTracker.NET is another free alternative, much simpler than Redmine but quite functional for small projects.

I am going to compare these systems considering our own practice of their usage by the following criteria:

• price;
• user interface language;
• web interface availability;
• lifecycle setup availability;
• authentication mechanisms supported;
• e-mail integration;
• reports creation availability;
• source control integration;
• user-defined fields;
• database management system (DBMS) used.

And here is the comparison table itself (Figure 1)

Below there are some comments on the values in the cells of the comparison table:

• Redmine and BugTracker.NET cost $0 because they are open source and free software that is their advantage (TCO minimization, further development by the interested-in community and your own resources) as well as their flaw (the community can fix found defects slowly or even ignore them while this could be critical in case you have now your own qualified resources).

• Redmine is the only system among the four that has a multi-lingual user interface. Some may say, software developers usually working in English environment do not really need a non-English interface of such a system (by the way, Redmine's interface is not translated completely into some languages). However, it is important because often non-IT people (customers, managers, etc.) become authors of new enhancements/tasks and sometimes even defects.

• Redmine and BugTracker.NET are web applications originally, and nowadays for such systems it is not only convenient but also necessary. ClearQuest was used as mostly a desktop application though had a web interface originally (unfortunately, based on Java that made impossible saving web pages as local HTML pages). VSTS was also used mostly a desktop application (Team Explorer in Visual Studio) but as a third-party web interface appeared (TeamPlain, later bought by Microsoft and renamed into VSTS Web Access) it became more popular.
• Lifecycle setup feature (when a special matrix define all possible transitions from one state to another for all roles) allows to reach better manageability and control over access rights. However, in small, self-organized teams it is enough to have a possibility to turn any state into any other one (of course, if the current role is given write permissions) as it is done in BugTracker.NET.

• Let all fans of Linux-like operating systems (non-Microsoft ones) put shame on me but I consider Windows authentication (Windows Active Directory, WAD) as a good and convenient thing, especially remembering that Windows clients and domains are still dominating in the world. For example, Redmine is described as supporting WAD authentication but it was not easy to make it working. Of course, in the ideal case we must talk about more general LDAP authentication, however, it definitely must be better than having a system's own user database ("one more username and password to remember").

• E-mail notifications essentially decrease the response time of all team members and improve the efficiency of each member individually and the team as a whole. The best implementation of this feature belongs to ClearQuest where the detailed notification scheme can be set, as well as notification letter templates. BugTracker.NET has a good implementation because you can subscribe to all needed changes, and though the fixed letter format is redundant it contains whatever you may think of. VSTS and Redmine's e-mail notifications are satisfactory. The latter has a mysterious notification scheme we still have not understood clearly. VSTS

has a poor built-in notification mechanism (a little bit improved by TeamPlain), and a fixed and poor notification letter format; however, everything becomes much better if you use the accompanying web interface called TeamAlerts.

• A sort of reporting (including search results and saved queries) presents in all of the compared systems (though I was complained of the constraints of Redmine's built-in reporting). Thanks to web interface there is a capability to save reporting results as local HTML pages in almost all systems (except ClearQuest which web interface is built on "unsavable" Java applets). "Exotic" reporting featured include: Remine's Atom (RSS) channels, and the ability to create agile custom reports by SQL Server Reporting Services directly connected to VSTS' database.

• Though source control integration is a doubtful advantage in a general case (particularly, when a change management system is accessible from the outside and is used by clients and customers), it is quite convenient for software developers (regarding linking recorded changes to source code versions and changes). Probably the most widespread source control system is Subversion (SVN) – that is why it is by default supported in Redmine and BugTracker.NET. However, when we used VSTS in one of our projects, we succeeded in integrating with Subversion as well. Meanwhile, ClearQuest insists on using its own (Rational) source control system, ClearCase.

• All the described systems have the ability to add your own (custom) fields to artifacts (we did not used them practically only in BugTracker.NET). Known restrictions include: Redmine does not allow arithmetic operations over custom fields (at least, without developing plugins); BugTracker.NET allows only 3 custom fields that look like drop-down lists with predifined values.

• The used database defines the infrastructural convenience (some database server may already be used in your intranet, while another may not) and particularly the total cost (Total Cost of Ownership, TCO) of the change management system (because proprietary database servers may cost much more that the system itself, especially if it is free). The leader by the number of supported database servers is ClearQuest; practically we used only SQL Server among the listed. By the way, SQL Server is the most popular among all database servers supported by the four change management systems. VSTS is "hard-coded" to use the "heavy" SQL Server while BugTracker.NET, as far as I know, can work with the free SQL Server Express. Redmine is the leader by the number of free database servers supported.

That's all folks! Didn't you expect I would tell you the global conclusion? :-) No, the conclusion is quite simple: everything depends on the context, i. e. the choice depends on the specific tasks, team members and, of course, the budget!.. Although I mentioned the budget only now, in fact in most cases it defines the selection of a system (by the way, not only a change management system). ■

Product Qualities Approach, Agile Style

intermediate

Author: ***Ryan Shriver***

About the author:

Ryan Shriver is a Managing Consultant with Dominion Digital, a Virginia-based process and technology-consulting firm. Based in Richmond, he leads the IT Performance Improvement Solution which includes Agile Adoption, Agile Engineering, IT Process Improvement and IT Services Management. With a background in systems architecture and large-scale agile development, Ryan currently focuses on measurable business value and systems engineering. He writes and speaks on these topics in the US and Europe, posting his current thoughts at theagileengineer.com.
Ryan can be reached at
rshriver@dominiondigital.com

Introduction

In many agile organizations, the product owner is responsible for setting the team's priorities through the product backlog. Whether they want enhancements to in-house systems or shrink-wrapped products, product owners get input from customers and stakeholders to create product backlogs of prioritized features (or user stories). These backlogs contain functionality that can be estimated by developers and planned for releases.

While there's nothing wrong with this approach of functions-first planning, I have come to believe it's short-sighted in that it doesn't place product qualities on equal pairing with functions. Currently in the agile community, there's a tendency to focus too quickly on user-centric functionality instead of product qualities that can deliver real stakeholder value, often very quickly. Product owners who understand and leverage product qualities cannot only delight customers, but also help them achieve their organization's business objectives.

This article provides a how-to for progressive change agents interested in delivering products that generate measurable business value for their customers and stakeholders. You'll learn how product qualities differ from functions, how to identify the right ones, measure them and use improvements to drive business results. Along the way, I'll demonstrate how to integrate an agile development processes such as Scrum.
What are Product Qualities?

Whereas functions describe what a product does, product qualities describe how well the product performs. This can be along an array of technical and business dimensions.

• Technical dimensions refer to how well the system performs, often referred to as "non-functional requirements". Common ones include availability, response time, throughput, storage capacity, security, maintainability and accuracy.

• Business dimensions refer to how well value is delivered to the stakeholders--the business results of the product. This includes market-facing product qualities important to paying customers as well as those related to operational objectives important to business sponsors.

How long does it take to record a business event? How much training is required for new hires? How much will our efficiency improve with your product? This article primarily explores the business dimensions, but the concepts are equally applicable to both, as you'll see.

Why are they important?

In crowded marketplaces where competitors have almost identical functions, products that perform at higher quality levels differentiate themselves from competitors. They also tend to be sold based upon their value propositions rather than viewed as a commodity, thus resulting in higher profit margins for the seller. Performing at higher-quality levels also has the benefit of being recognized as a leader in your industry, something that can only help sales.

For software, desired product qualities are commonly financially driven: increase revenues and reduce costs. Yet some product qualities can also have non-financial objectives such as customer satisfaction, net promoter score and team morale. Often, these nonfinancial objectives are important leading indicators of future financial results and are thusly important to consider.

One organization that puts product qualities forefront in their product development approach is Confirmit. They sell marketing research software and report their products-focused approach is one of their keys to success. Another organization that markets using product qualities is Unica, the marketing software vendor. While I've never used their products, I do think they market quite well using product qualities aligned with their customer's needs. These include generating higher sales, retaining more customers and reducing operating costs. Notice these product qualities aren't features

or functions, they are the business objectives of their customer: marketing department executives.

Here's a graphic from their website on the value proposition of their products:

Unica must do well enough with their approach to marketing. Gartner has it at the top of their magic quadrant for multi-channel campaign management, ahead of SAS, Teradata, Oracle-Siebel and other industry leaders.

Identifying the Right Product Qualities

So how do we identify the right product qualities? While I don't believe there's a single right way for everyone, I've found the following recipe works well for me. I encourage you to explore what works best for you:

1. Identify the Product Stakeholders. Pair up with a partner and identify all the possible product stakeholders you can think of. Cast a wide net and identify anyone who is impacted by the product. Often the objectives of your stakeholders make good product qualities! Key stakeholder types include:

• Customer – purchases your product,
• Business Sponsor – funds development of the product
• User – various roles, uses your product to accomplish a task
• Operations – provides infrastructure and servicing
• Trainers – trains new users on your product

2. Identify the Product Qualities. Organize meetings with the individual stakeholders to learn about their impact on the product.

Come prepared with a set of draft product qualities and center the discussions on the important aspects of the product to them. I've found good questions that help reveal product qualities include:
• What are the reasons customers purchase your product?
• What are your customer's objectives and how does your product help them achieve these?
• What would it mean to you personally, the organization and your customers if <insert product quality> improved?

3. Build Consensus. Organize a workshop and present the key stakeholders the information you've gathered. Encourage candid discussion, but work toward getting consensus on three things:
• Highest priority stakeholder to serve first
• Highest priority product qualities to improve first
• Available budget of time and money for next release
Be efficient with stakeholder's time but also flexible to explore certain areas for deeper discussions. The goal is to get consensus to the highest priority product qualities for improvement next--not forever. There will be time to re-prioritize later based upon feedback; all we're looking for is a starting place for improvement. Ideally we prioritize all the product qualities, but if that's not possible, identifying the most important one is better than none at all.

You'll also want to do this same exercise with the technical leads for system qualities such as availability and security. The discussions should be around design ideas necessary to avoid the constraint levels--and ideally hit the target levels within the budgeted resources. If this cannot be done, the technical team's responsibility is to come back with alternative target or constraint levels that are achievable in budget, and use this for further stakeholder education and discussion. Depending on your resources, you can do this sequentially or in parallel to identifying product qualities.

These prioritized product qualities can go into your results backlog and serve as input into your agile release planning process. As to estimating how much these products will improve in the next release, we need to learn how to measure our product qualities.

Measurable Product Qualities

While there's value in simply identifying and prioritizing the product qualities, the primary goal is to measure them. Why? Tom Gilb says it best:

"The fact that we can set numeric objectives, and track them, is powerful; but in fact is not the main point. The main purpose of quantification is to force us to think deeply, and debate exactly, what we mean; so that others, later, cannot fail to understand us."

Defining measurable levels of improvement to product qualities forces us to have open and honest discussion with stakeholders. This ensures expectations are aligned with how much better, faster, quicker or improved the new release will be while working within the resources - or else what additional resources are necessary to reach the desired levels.

I prefer to define product qualities with the following minimum attributes:
• **Name** – Brief unique identifier
• **Scale** – What's measured (units)
• **Meter** – How it's measured (method)
• **Targets** – Levels aiming to achieve
• **Constraints** – Levels trying to avoid
• **Benchmark** – Current or past performance levels
In order to fill in the details for the highest priority system qualities, I use a similar approach to before. Working with my partner, we create a draft set based on our working knowledge and then validate and fill in the gaps with stakeholders offline. We then gather stakeholders together again to present the information back and get consensus on each attribute. Depending on your project, you may find it more efficient to do a single workshop to accomplish all of this. The method used to gather the product qualities isn't as important as ensuring the information is complete and that consensus is reached amongst key stakeholders.

The following figures illustrate an example product company seeking to increase market share, monetary donations and volunteer time donations. Below are the product qualities most important to the business sponsors.

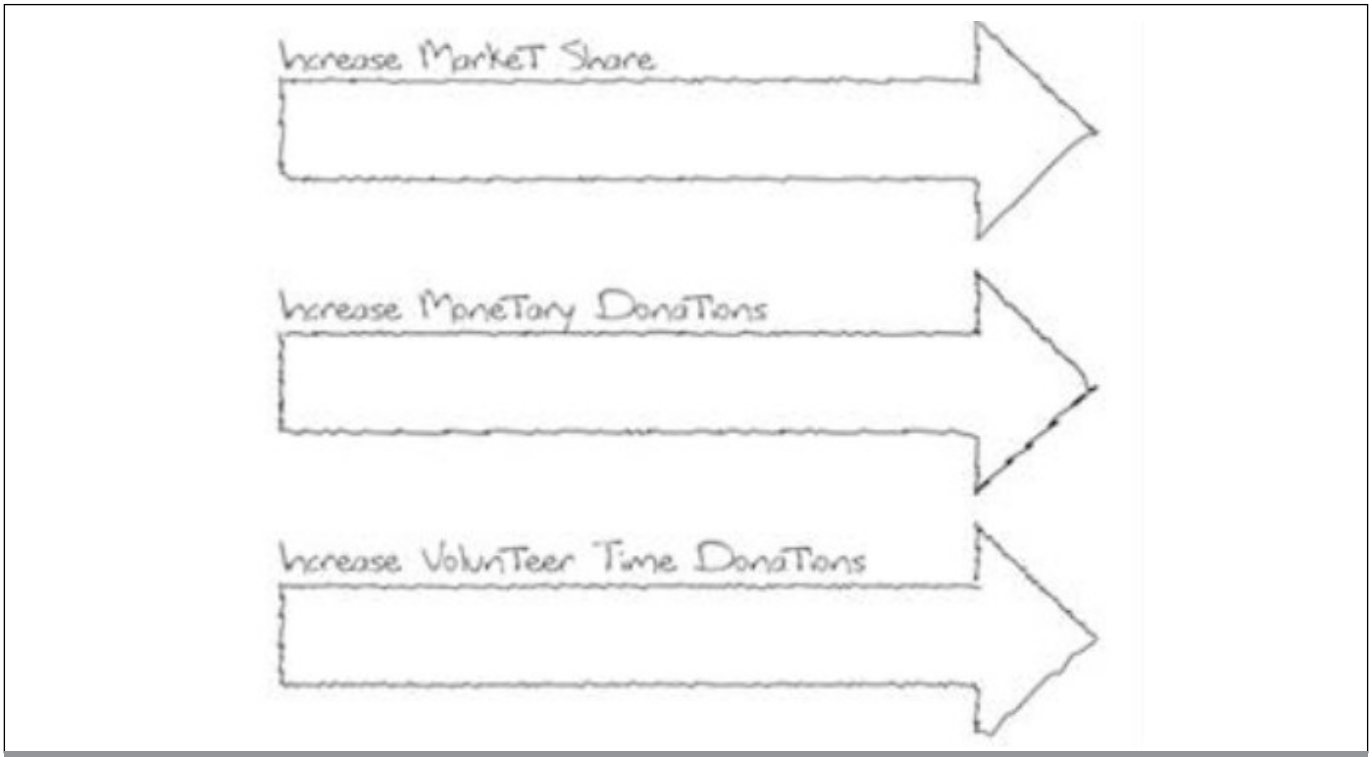


Figure 1 Product Qualities

Figure 2 With Scale and Meter added

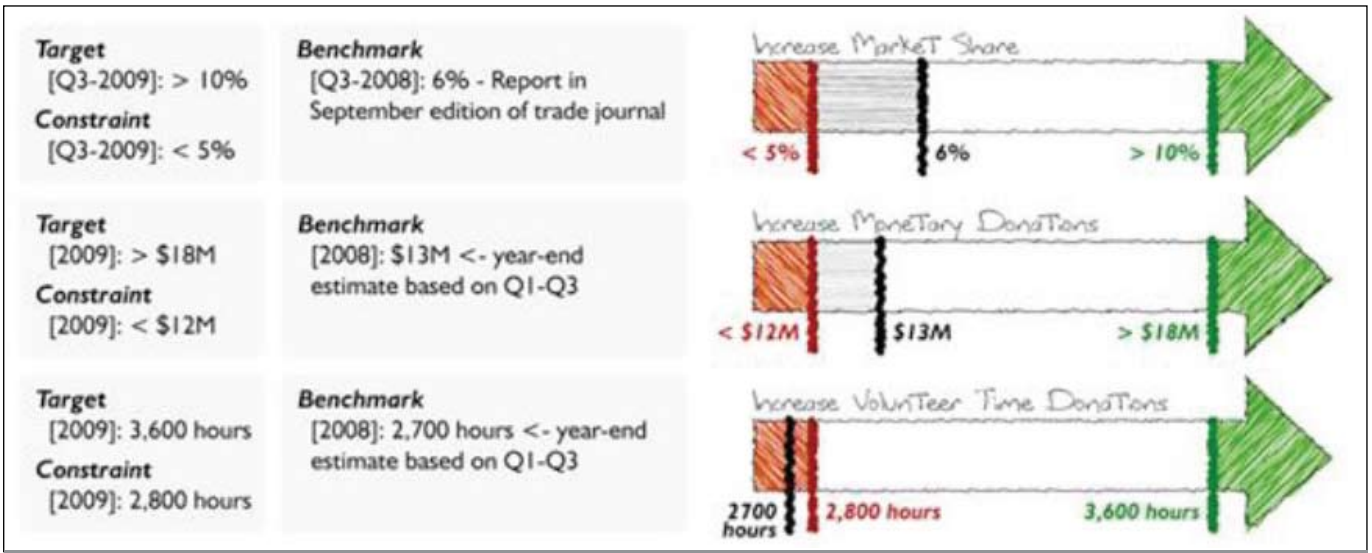


Figure 3 Completed with Target, Constraint and Benchmark

Let's look at an example system quality using the same attributes:

Name: Availability
Scale: Percentage of time system is available for accepting transactions excluding 1 hour weekly maintenance window
Meter: Weekly Monitoring Report from data center polling application every 3 minutes
Target: >= 99.9% (< 10 mins unplanned down)
Constraint: < 99.4% (> 60 mins unplanned down)
Benchmark [Date/Version]: ___% <- Benchmark Source

Hopefully you can see from these examples that product qualities can be defined simply and succinctly. (In my experience, three product qualities can fit on one PowerPoint slide and up to 10 on a piece of paper!)

Planning and Reporting with Product Qualities

Now that we've quantified our product qualities, they can be integrated into planning and reporting activities such as:

Balanced Scorecard – For the executives, integrating product qualities reporting into the scorecards can clearly communicate progress toward targets on the highest priority product qualities (including what resources were used to achieve these results). Figure 3 above shows one visual representation of progress toward targets useful for reporting to executives (but often text is the simplest and easiest means to communicate results).

Results Backlog – In my recent article, I discussed product owners using a companion to the Scrum product backlog called a results backlog. The idea was to create an artifact to manage and prioritize business objectives so the actual business results, in addition to the product features, could be managed. The results backlog concept applies to the product and system qualities as well because both measure ends, not means.

Release Planning – For the product owner, agile coach and the team, as you evaluate each new proposed feature during release planning, ask yourself:
• Which of our highest priority product qualities will this feature improve?
• Will this feature alone get us to our target performance level or do we need to consider additional or alternative features, too?
• What percentage of our resources (time and money) will it take to implement this feature (and what's remaining to improve other quality levels)?

Although these questions can work at the user-story level, I find it's helpful to work at the feature level first -before breaking the feature down into user stories. If a feature has positive impacts on the product qualities, then the component user stories should as well.

Value Decisions– Sometimes referred to as an impact estimation table, a

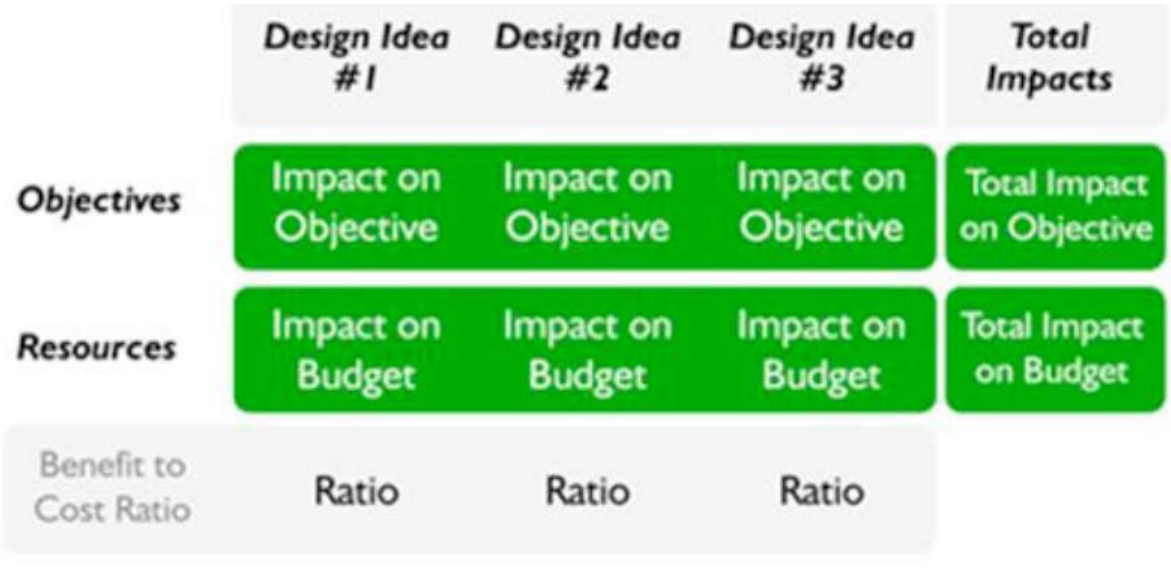


Figure 4 Value Decision Table

value decision table helps make informed decisions by assessing how means (such as technical design ideas, features or projects) impact ends (such as product qualities or business objectives) using some percentage of the budgeted resources (see Figure 4). The result is a benefit-to-cost ratio that indicates the "bang for the buck" delivered. In addition, by summing the impacts we can see the total impacts if all design ideas were implemented.

For an example of how this can be integrated with Scrum, see my article Measurable Value with Agile.

Deliver, Review, Adjust and Repeat

While much of this article has been about planning, there's still execution that must excel in order to achieve results. While I don't overlook this aspect, I have learned that well-coached agile teams can start delivering software on a frequent basis relatively quickly. While I know from first-hand experience that software delivery is challenging in its own right, I've learned that most agile organizations learn how to do the thing right early on but can struggle with knowing how to do the right thing for years!

It is important to review progress toward goals at key milestones such as each release or monthly or quarterly meetings. Pay special attention to the resources necessary to reach the performance levels. Does this reveal a new insight? Does this set (or reset) expectations on what's realistically achievable in the future? If the technical team was over confident the last time, now might be the time to lower target levels or increase budget in order to reach target levels. These are all topics ideal for discussion. Remember: The goal is to continually improve and adjust as you go, reviewing at the right times to make informed decisions.

How this works in one particular organization will vary from another, but the important point is to do it repeatedly in order to gain the benefits of learning and continuous improvement.

Summary

Today we've learned what product qualities are, why they are important and how to identify, prioritize and quantifying them succinctly. We've learned how to make better-informed decisions with numbers and communicate results to stakeholders.

Together, these techniques form the basis of a product qualities approach to development that can be integrated with agile development teams. This approach can help your agile teams focus on what's most important to the stakeholders using clear terms and numbers everyone understands. The result is a win-win: Stakeholders get measurable results on their highest priorities and the team gets the satisfaction to knowing they are making a real difference.

Reference

The URL for this article is: http://www.gantthead.com/article.cfm?ID=254127

What's fundamentally wrong?

Improving our approach towards capturing value in requirements specification

advanced

Author: ***Tom Gilb and Lindsay Brodie***

About the authors:

Tom is the author of nine books, and hundreds of papers on these and related subjects. His latest book 'Competitive Engineering' is a substantial definition of requirements ideas. His ideas on requirements are the acknowledged basis for CMMI level 4 (quantification, as initially developed at IBM from 1980). Tom has guest lectured at universities all over UK, Europe, China, India, USA, Korea – and has been a keynote speaker at dozens of technical conferences internationally.

www.gilb.com, twitter: @imTomGilb

Lindsey Brodie is currently carrying out research on prioritization of stakeholder value, and teaching part-time at Middlesex University. She has an MSc in Information Systems Design from Kingston Polytechnic. Her first degree was Joint Honours Physics and Chemistry from King's College, London University. Lindsey worked in industry for many years, mainly for ICL. Initially, Lindsey worked on project teams on customer sites (including the Inland Revenue, Barclays Bank, and J. Sainsbury's) providing technical support and developing customised software for operations. From there, she progressed to product support of mainframe operating systems and data management software: databases, data dictionary and 4th generation applications. Having completed her Masters, she transferred to systems development - writing feasibility studies and user requirements specifications, before working in corporate IT strategy and business process re-engineering.

Lindsey has collaborated with Tom Gilb and edited his book, "Competitive Engineering". She has also co-authored a student textbook, "Successful IT Projects" with Darren Dalcher (National Centre for Project Management). She is a member of the BCS and a Chartered IT Practitioner (CITP).

Abstract

We are all aware that many of our IT projects fail and disappoint: the poor state of requirements practice is frequently stated as a contributing factor. This article proposes a fundamental cause is that we think like programmers, not engineers and managers. We fail to concentrate on value delivery, and instead focus on functions, on use-cases and on code delivery. Our requirements specification practices fail to adequately address capturing value-related information. Compounding this problem, senior management is not taking its responsibility to make things better: managers are not effectively communicating about value and demanding value delivery. This article outlines some practical suggestions aimed at tackling these problems and improving the quality of requirements specification.

Keywords: Requirements; Value Delivery; Requirements Definition; Requirements Specification

Introduction

We know many of our IT projects fail and disappoint, and that the overall picture is not dramatically improving [1] [2]. We also know that the poor state of requirements practice is frequently stated as one of the contributing failure factors [3] [4]. However, maybe a more fundamental cause can be proposed? A cause, which to date has received little recognition, and that certainly fails to be addressed by many well known and widely taught methods. What is this fundamental cause? In a nutshell: that we think like programmers, and not as engineers and managers. In other words, we do not concentrate on value delivery, but instead focus on functions, on use cases and on code delivery. As a result, we pay too little attention to capturing value and value-related information in our requirements specifications. We fail to capture the information that allows us to adequately consider priorities, and engineer and manage stakeholder-valued solutions.

This article outlines some practical suggestions aimed at tackling these problems and improving the quality of requirements specification. It focuses on 'raising the bar' for communicating about

value within our requirements. Of course, there is much still to be learnt about specifying value, but we can make a start – and achieve substantial improvement in IT project delivery – by applying what is already known to be good practice.

Note there is little that is new in what follows, and much of what is said can be simply regarded as commonsense. However, since IT projects continue not to grasp the significance of the approach advocated, and as there are people who have yet to encounter this way of thinking, it is worth repeating!

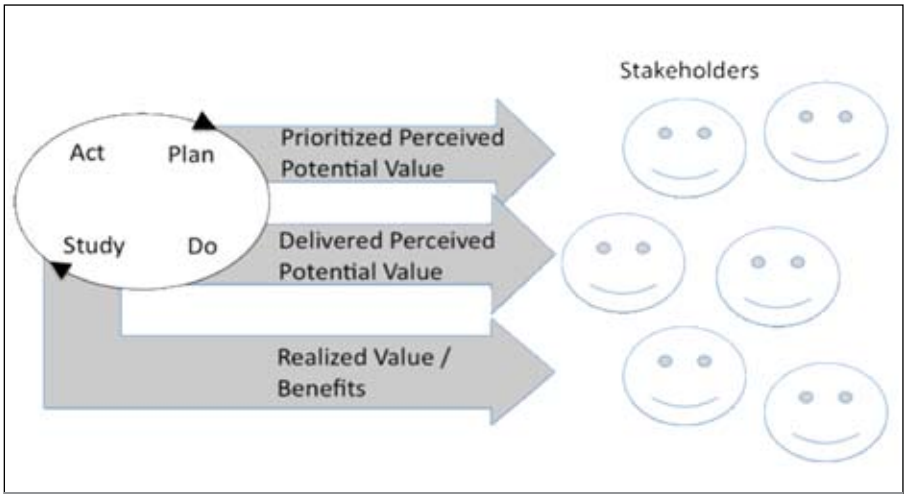


Figure 1 Value can be delivered gradually to stakeholders. Different stakeholders will perceive different value.

Definition of Value

The whole point of a project is achieving 'realized value' (also known as 'benefits'), for the stakeholders: it is not the defined functionality, and not the user stories that actually count. Value can be defined as 'the benefit we think we get from something' [5, page 435]. See Figure 1.

Notice the subtle distinction between initially perceived value ('I think that would be useful'), and realized value: effective and factual value ('this was in practice more valuable than we thought it would be, because …'). Realized value has dependencies on the stakeholders actually utilizing a project's deliverables.

The issue with much of the conventional requirements thinking is that it is not closely enough coupled with 'value'. IT business analysts frequently fail to gather the information supporting a more precise understanding and/or the calculation of value. Moreover, the business people when stating their requirements frequently fail to justify them using value. The danger if requirements are not closely tied to value is that we lack the basic information allowing us to engineer and prioritize implementation to achieve value delivery, and we risk failure to deliver the required expected value, even if the 'requirements' are satisfied.

It is worth pointing out that 'value' is multi-dimensional. A given requirement can have financial value, environmental value, competitive advantage value, architectural value, as well as many other dimensions of value. Certainly value requires much more explicit definition than the priority groups used by MoSCoW ('Must Have', 'Should Have', 'Could Have', and 'Would like to Have/Won't Have This Time') [6] or by the Planning Game ('Essential', 'Less Essential' and 'Nice To Have') [7] for prioritizing requirements. Further, for an IT project, engineering 'value' also involves consideration of not just the requirements, but also the optional designs and the resources available: tradeoffs are needed. However, these are topics for future articles, this article focuses on the initial improvements needed in requirements specification to start to move towards value thinking.

Definition of Requirement

Do we all have a shared notion of what a 'requirement' is? This is another of our problems. Everybody has an opinion, and many of the opinions about the meaning of the concept 'requirement' are at variance: few of the popular definitions are correct or useful - especially when you consider the concept of 'value' alongside them. We have decided to define a requirement as a "stakeholder-valued end state". You possibly will not accept, or use this definition yet, but we have chosen it to emphasize the 'point' of IT systems engineering.

In previous work, we have identified, and defined a large number of requirement concepts [5, see Glossary, pages 321-438]. A sample of these concepts is given in Figure 2. You can use these concepts and the notion of a "stakeholder-valued end state" to re-examine your current requirements specifications. In the rest of this article, we provide more detailed discussion about some of the key points (the "key principles") you should consider.

The Key Principles

The key principles are summarized in Figure 3. Let's now examine these principles in more detail.

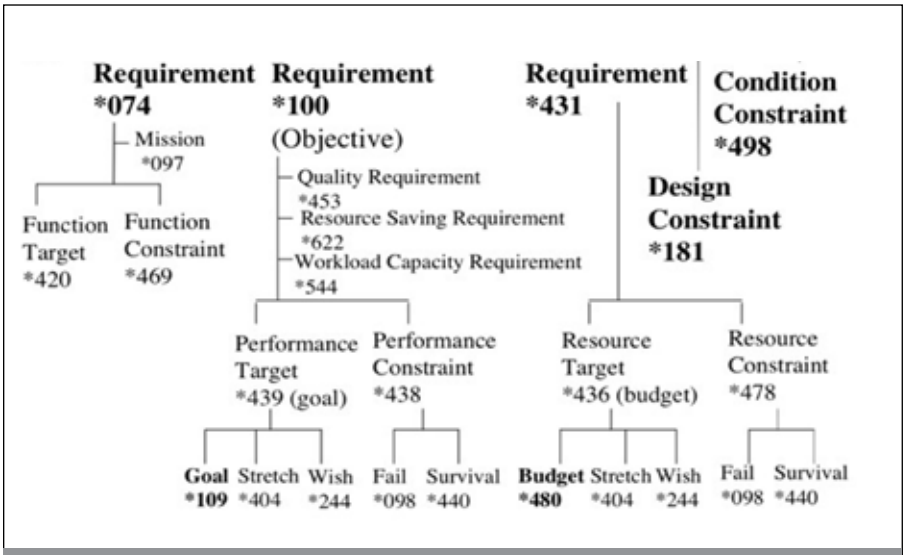


Figure 2 Example of Planguage requirements concepts

Note, unless otherwise specified, further details on all aspects of Planguage (a planning language developed by one of the authors, Tom Gilb) can be found in [5].

Ten Key Principles for Successful Requirements

1. Understand the top level critical objectives

2. Think stakeholders: not just users and customers!

3. Focus on the required system quality, not just its functionality

4. Quantify quality requirements as a basis for software engineering

5. Don't mix ends and means

6. Capture explicit information about value

7. Ensure there is 'rich specification': requirement specifications need far more information than the requirement itself!

8. Carry out specification quality control (SQC)

9. Consider the total lifecycle and apply systems-thinking - not just a focus on software

10. Recognize that requirements change: use feedback and update requirements as necessary

Figure 3 Ten Key Principles for Successful Requirements

Principle 1. Understand the top-level critical objectives

The 'worst requirement sin of all' is found in almost all the IT projects we look at, and this applies internationally. Time and again, the high-level requirements – also known as the top-level critical objectives (the ones that fund the project), are vaguely stated, and ignored by the project team. Such requirements frequently look like the example given in Figure 4 (which has been slightly edited to retain anonymity). These requirements are for a real project that ran for eight years and cost over 100 million US dollars. The project failed to deliver any of them. However, the main problem is that these are not top-level critical objectives: they fail to explain in sufficient detail what the business is trying to achieve: there are no real pointers to indicate the business aims and priorities. There are additional problems as well that will be discussed further later (such as lack of quantification, mixing optional designs into the requirements, and insufficient background description).

Example of Initial Weak Top-Level Critical Objectives

1. Central to the corporation's business strategy is to be the world's premier integrated <domain> service provider

2. Will provide a much more efficient user experience

3. Dramatically scale back the time frequently needed after the last data is acquired to time align, depth correct, splice, merge, recomputed and/or do whatever else is needed to generate the desired products

4. Make the system much easier to understand and use than has been the case with the previous system

5. A primary goal is to provide a much more productive system development environment then was previously the case

6. Will provide a richer set of functionality for supporting next generation logging tools and applications

7. Robustness is an essential system requirement

8. Major improvements in data quality over current practices

Figure 4 Example of Initial Weak Top Level Critical Objectives

Management at the CEO, CTO and CIO level did not take the trouble to clarify these critical objectives. In fact, the CIO told me that the CEO actively rejected the idea of clarification! So management lost control of the project at the very beginning. Further, none of the technical 'experts' reacted to the situation. They happily spent $100 million on all the many suggested architecture solutions that were mixed in with the objectives.

It actually took less than an hour to rewrite one of these objectives, "Robustness", so that it was clear, measurable, and quantified (see later). So in one day's work the project could have clarified the objectives, and perhaps avoided some of the eight years of wasted time and effort.

Principle 2. Think stakeholders: not just users and customers!

Too many requirements specifications limit their scope to being too narrowly focused on user or customer needs. The broader area of stakeholder needs and values should be considered, where a 'stakeholder' is anyone or anything that has an interest in the system [5, page 420]. It is not just the users and customers that must be considered: IT development, IT maintenance, senior management, operational management, regulators, government, as well as other stakeholders can matter. The different stakeholders will have different viewpoints on the requirements and their associated value. Further, the stakeholders will be "experts" in different areas of the requirements. These different viewpoints will potentially lead to differences in opinion over the implementation priorities.

Principle 3. Focus on the required system quality, not just its functionality

Far too much attention is paid to what the system must do (function) and far too little attention is given to how well it should do it (qualities). Many requirements specifications consist of detailed explanation of the functionality with only brief description of the required system quality. This is in spite of the fact that quality improvements tend to be the major drivers for new projects.

In contrast, here's an example, the Confirmit case study [8], where the focus of the project was not on functionality, but on driving up the system quality. By focusing on the "Usability" and "Performance" quality requirements the project achieved a great deal! See Table 1.

Description of requirement/work task	Past	Current Status
Usability.Productivity: Time for the system to generate a survey	7200 sec	15 sec
Usability.Productivity: Time to set up a typical market research report	65 min	20 min
Usability.Productivity: Time to grant a set of end-users access to a report set and distribute report login information	80 min	5 min
Usability.Intuitiveness: The time in minutes it takes a medium-experienced programmer to define a complete and correct data transfer definition with Confirmit Web Services without any user documentation or any other aid	15 min	5 min
Performance.Runtime.Concurrency: Maximum number of simultaneous respondents executing a survey with a click rate of 20 sec and a response time < 500ms given a defined [Survey Complexity] and a defined [Server Configuration, Typical]	250 users	6000

Table 1 Extract from Confirmit Case Study [8]

By system quality we mean all the "-ilities" and other qualities that a system can express. Some system developers limit system quality to referring to bug levels in code. However, a broader definition should be used. System qualities include availability, usability, portability, and any other quality that a stakeholder is interested in, like intuitiveness or robustness. See Figure 5, which shows a set of quality requirements. It also shows the notion that resources are "input" or used by a function, which in turn "outputs" or expresses system qualities. Sometimes the system qualities are mis-termed "non-functional requirements (NFRs)", but as can be seen in this figure, the system qualities are completely linked to the system functionality. In fact, different parts of the system functionality are likely to require different system qualities.

Figure 5 A way of visualizing qualities in relation to function and cost. Qualities and costs are scalar variables, so we can define scales of measure in order to discuss them numerically. The arrows on the scale arrows represent interesting points, such as the requirement levels. The requirement is not 'security' as such, but a defined, and testable degree of security [5, page 163]

Principle 4. Quantify quality requirements as a basis for software engineering

Frequently we fail to practice "software engineering" in the sense of real engineering as described by engineering professors, like Koen [9]. All too often quality requirements specifications consist merely of words. No numbers, just nice sounding words; good enough to fool managers into spending millions for nothing (for example, "a much more efficient user experience").

We seem to almost totally avoid the practice of quantifying qualities. Yet we need quantification in order to make the quality requirements clearly understood, and also to lay the basis for measuring and tracking our progress in improvement towards meeting them. Further, it is the quantification that is the key to a better understanding of cost and value – different levels of quality have different associated cost and value.

The key idea for quantification is to define, or reuse a definition, of a scale of measure. For example, for a quality "Intuitiveness", a sub-component of "Usability":

To give some explanation of the key quantification features in Figure 6:

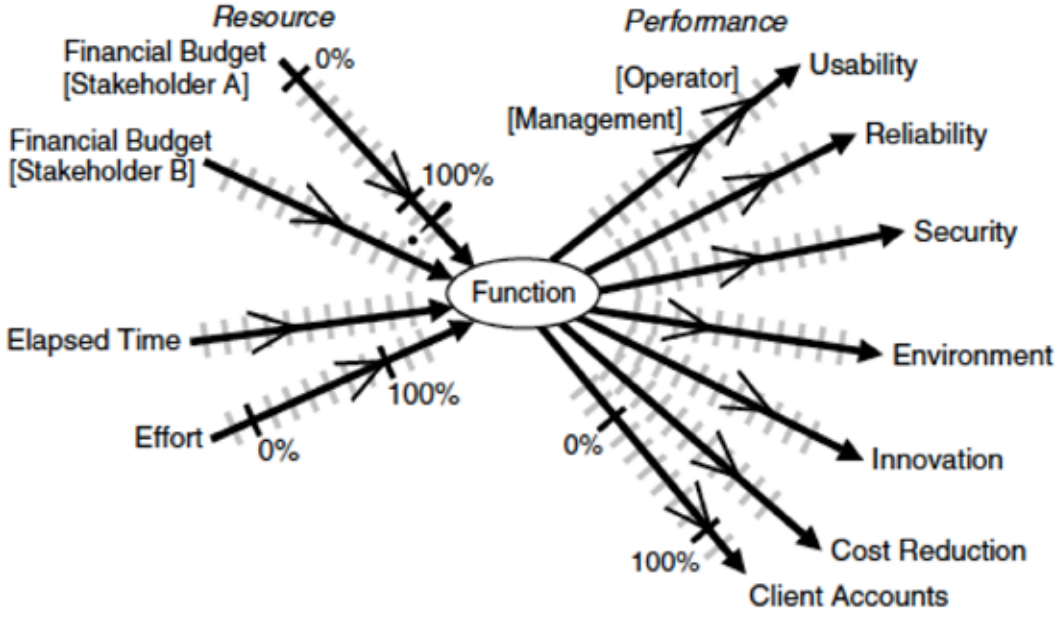


Figure 5

Usability.Intuitiveness:

Type: Marketing Product Quality Requirement.
Ambition: Any potential user, any age, can immediately discover and correctly use all functions of the product, without training, help from friends, or external documentation.
Scale: % chance that defined [User] can successfully complete defined [Tasks] <immediately> with no external help.
Meter: Consumer reports tests all tasks for all defined user types, and gives public report.
Goal [Market = USA, User = Seniors, Product = New Version, Task = Photo Tasks Set, When = 2012]: 80% ±10% <- Draft Marketing Plan.

Figure 6 A simple example of quantifying a quality requirement, 'Intuitiveness'.

1. Ambition is a high-level summary of the requirement. One that is easy to agree to, and understand roughly.

2. Scale is the formal definition of our chosen scale of measure. The parameters [User] and [Task] allow us to generalize here, while becoming more specific in detail below (see later). They also encourage and permit the reuse of the Scale, as a sort of 'pattern'.

3. Meter provides a defined measuring process. There can be more than one for different occasions.

4. Goal is one of many possible requirement levels (see earlier detail in Figure 2 for some others: Stretch, Wish, Fail and Survival). We are defining a stakeholder-valued future state (for example: 80% ± 10%).

One stakeholder is 'USA Seniors'. The future is 2012. The requirement level type, Goal, is defined as a very high priority, budgeted promise of delivery. It is of higher priority than a Stretch or Wish level. Note other priorities may conflict and prevent this particular requirement from being delivered in practice.

If you know the *conventional* state of requirements methods, then you will now, from this example alone, begin to appreciate the difference proposed by such quantification - especially for *quality* requirements. IT projects already quantify time, cost,, response time, burn rate, and bug density – but there is much more to *achieve system engineering!*

Here is another example of quantification (see Figure 7). It is the initial stage of the rewrite of Robustness from the Figure 4 example. First we determined that Robustness is complex and composed of many different attributes, such as Testability.

Robustness:

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

Figure 7 Definition of a complex quality requirement, Robustness

Then we defined Testability in more detail (see Figure 8).

Testability:

Type: Software Quality Requirement.
Version: Oct 20, 2006.
Status: Draft.
Stakeholder: {Operator, Tester}.
Ambition: Rapid duration automatic testing of <critical complex tests> with extreme operator setup and initiation.
Scale: The duration of a defined [Volume] of testing or a defined [Type of Testing] by a defined [Skill Level] of system operator under defined [Operating Conditions].
Goal [All Customer Use, Volume = 1,000,000 data items, Type of Testing = WireXXXX vs. DXX, Skill Level = First Time Novice, Operating Conditions = Field]: < 10 minutes.
Design: Tool simulators, reverse cracking tool, generation of simulated telemetry frames entirely in software, application specific sophistication for drilling – recorded mode simulation by playing back the dump file, application test harness console <- 6.2.1 HFS.

Figure 8 Quantitative definition of Testability, an attribute of Robustness

Note this example shows the notion of there being different levels of requirements. Principle 1 also has relevance here as it is concerned with top-level objectives (requirements). The different levels that can be identified include: corporate requirements, the top-level critical few project or product requirements, system requirements and software requirements. We need to clearly document the level and the interactions amongst these requirements.

An additional notion is that of 'sets of requirements'. Any given stakeholder is likely to have a set of requirements rather than just an isolated single requirement. In fact, achieving value could depend on meeting an entire set of requirements.

Principle 5. Don't mix ends and means

"Perfection of means and confusion of ends seem to characterize our age." Albert Einstein. 1879-1955
The problem of confusing ends and means is clearly an old one, and deeply rooted. We specify a solution, design and/or architecture, instead of what we really value – our real requirement. There are explanatory reasons for this – for example solutions are more concrete, and what we want (qualities) are more abstract for us (because we have not yet learned to make them measurable).

The problems occur when we do confuse them: if we do specify the means, and not our true ends. As the saying goes: "Be careful what you ask for, you might just get it" (unknown source). The problems include:
• You might not get what you really want
• The solution you have specified might cost too much or have bad side effects, even if you do get what you want
• There may be much better solutions you don't know about yet.

So how to we find the 'right requirement', the 'real requirement' [10] that is being 'masked' by the solution? Assume that there probably is a better formulation, which is a more accurate expression of our real values and needs. Search for it by asking 'Why?' Why do I want X, it is because I really want Y, and assume I will get it through X. But, then why do I want Y? Because I really want Z and assume that is the best way to get X. Continue the process until it seems reasonable to

stop. This is a slight variation on the '5 Whys' technique [11], which is normally used to identify root causes of problems (rather than high-level requirements).

Assume that our stakeholders will usually state their values in terms of some perceived means to get what they really value. Help them to identify (The 5 Whys?) and to acknowledge what they really want, and make that the 'official' requirement. Don't insult them by telling them that they don't know what they want. But explain that you will help them more-certainly get what they more deeply want, with better and cheaper solutions, perhaps new technology, if they will go through the '5 Whys?' process with you. See Figure 9.

Why do you require a 'password'? For Security!

What kind of security do you want? Against stolen information.

What level of strength of security against stolen information are you willing to pay for? At least a 99% chance that hackers cannot break in within 1 hour of trying! Whatever that level costs up to €1 million.

So that is your real requirement? Yep.

Can we make that the official requirement, and leave the security design to both our security experts, and leave it to proof by measurement to decide what is really the right design? Of course!

The aim being that whatever technology we choose, it gets you the 99%?

Sure, thanks for helping me articulate that!

Figure 9 Example of the requirement, not the design feature, being the real requirement

Note that this separation of designs from the requirements does not mean that you ignore the solutions/designs/architecture when software engineering. It is just that you must separate your requirements - including any mandatory means - from any optional means. The key thing is to understand what is optional so that you consider alternative solutions. See Figure 10, which shows two alternative solutions: Design A with Designs B and C, or Design A with Design D. Assuming that say, Design B was mandatory, could distort your project planning.

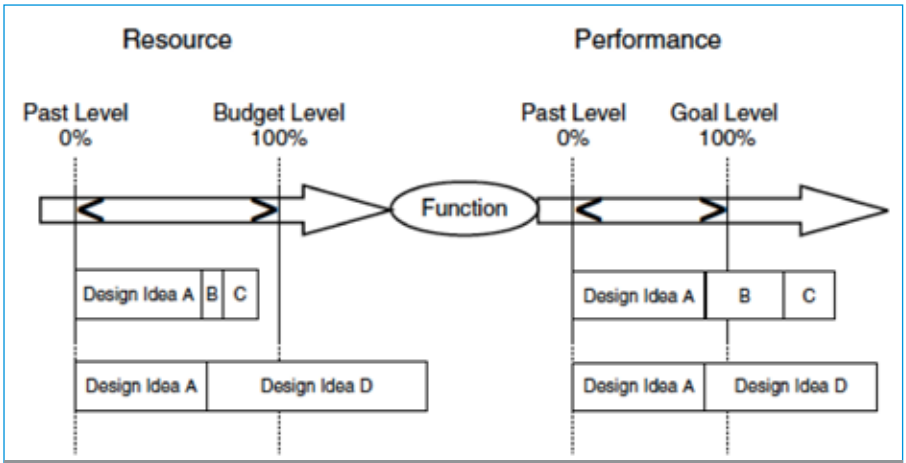


Figure 10

Figure 10 A graphical way of understanding performance attributes (which include all qualities) in relation to function, design and resources. Design ideas cost some resources, and design ideas deliver performance (including system qualities) for given functions.

Principle 6. Capture explicit information about value

How can we articulate and document notions of value in a requirement specification? See the example for Intuitiveness, a component quality of Usability, given in Figure 11, which expands on Figure 6.

Usability.Intuitiveness:

Type: Marketing Product Requirement.
Stakeholders: {Marketing Director, Support Manager, Training Center}.
Impacts: {Product Sales, Support Costs, Training Effort, Documentation Design}.
Supports: Corporate Quality Policy 2.3.
Ambition: Any potential user, any age, can immediately discover and correctly use all functions of the product, without training, help from friends, or external documentation.
Scale: % chance that a defined [User] can successfully complete the defined [Tasks] <immediately>, with no external help.
Meter: Consumer Reports tests all tasks for all defined user types, and gives public report.

Analysis

Trend [Market = Asia, User = {Teenager, Early Adopters}, Product = Main Competitor, Projection = 2013]: 95%±3% <- Market Analysis.
Past [Market = USA, User = Seniors, Product = Old Version, Task = Photo Tasks Set, When = 2010]: 70% ±10% <- Our Labs Measures.
Record [Market = Finland, User = {Android Mobile Phone, Teenagers}, Task = Phone+SMS Task Set, Record Set = January 2010]: 98% ±1% <- Secret Report.

Our Product Plans

Goal [Market = USA, User = Seniors, Product = New Version, Task = Photo Tasks Set, When = 2012]: 80% ±10% <- Draft Marketing Plan.
Value [Market =USA, User = Seniors, Product = New Version, Task = Photo Tasks Set, Time Period = 2012]: 2M USD.
Tolerable [Market = Asia, User = {Teenager, Early Adopters}, Product = Our New Version, Deadline = 2013]: 97%±3% <- Marketing Director Speech.
Fail [Market = Finland, User = {Android Mobile Phone, Teenagers}, Task = Phone+SMS Task Set, Product Release 9.0]: Less Than 95%.
Value [Market = Finland, User = {Android Mobile Phone, Teenagers}, Task = Phone+SMS Task Set, Time Period = 2013]: 30K USD.

Figure 11

Figure 11 A fictitious Planguage example, designed to display ways of making the value of a requirement clear

For brevity, a detailed explanation is not given here. Hopefully, the Planguage specification is reasonably understandable without detailed explanation. For example, the Goal statement (80%) specifies which market ("USA") and users ("Seniors") it is intended for, which set of tasks are valued (the "Photo Tasks Set"), and when it would be valuable to get it delivered ("2012"). This 'qualifier' information in all the statements, helps document where, who, what, and when the quality level applies. The additional Value parameter specifies the perceived value of achieving 100% of the requirement. Of course, more could be said about value and its specification, this is merely a 'wake-up call' that explicit value needs to be captured within requirements. It is better than the more common specifications of the Usability requirement, that we often see, such as: "The product will be more user-friendly, using Windows".

So who is going to make these value statements in requirements specifications? I don't expect developers to care much about value statements. Their job is to deliver the requirement levels that someone else has determined are valued. Deciding what sets of requirements are valuable is a Product Owner (Scrum) or Marketing Management function. Certainly, the IT staff should only determine the value related to IT stakeholder requirements!

Principle 7. Ensure there is 'rich specification': requirement specifications need far more information than the requirement itself!

Far too much emphasis is often placed on the requirement itself; and far too little concurrent information is gathered about its background, for example: who wants this requirement and when? The requirement itself might be less than 10% of a complete requirement specification that includes the background information. It should be a corporate standard to specify this related background information, and to ensure it is intimately and immediately tied into the requirement itself.

Such background information is useful related information, but is not central (core) to the implementation, and nor is it commentary. The central information includes: Scale, Meter, Goal, Definition and Constraint.

Background specification includes: benchmarks {Past, Record, Trend}, Owner, Version, Stakeholders, Gist (brief description), Ambition, Impacts, and Supports. The rationale for background information is as follows:

- To help judge the value of the requirement
- To help prioritize the requirement
- To help understand the risks associated with the requirement
- To help present the requirement in more or less detail for various audiences and different purposes
- To give us help when updating a requirement
- To synchronize the relationships between different but related levels of the requirements
- To assist in quality control of the requirements
- To improve the clarity of the requirement.

Commentary is any detail that probably will not have any economic, quality or effort consequences if it is incorrect, for example, notes and comments.

See Figure 12 for an example, which illustrates the help given by background information regarding risks.

Testability:

Type: Performance Quality.
Owner: Quality Director. Author: John Engineer.
Stakeholders: {Users, Shops, Repair Centers}.
Scale: Mean Time Between Failure.
Goal [Users]: 20,000 hours <- Customer Survey, 2004.
Rationale: Anything less would be uncompetitive.
Assumption: Our main competitor does not improve more than 10%.
Issues: New competitors might appear.
Risks: The technology costs to reach this level might be excessive.
Design Suggestion: Triple redundant software and database system.
Goal [Shops]: 30,000 hours <- Quality Director.
Rationale: Customer contract specification.
Assumption: This is technically possible today.
Issues: The necessary technology might cause undesired schedule delays.
Risks: The customer might merge with a competitor chain and leave us to foot the costs for the component parts that they might no longer require.
Design Suggestion: Simplification and reuse of known components.

Figure 12

Figure 12 A requirement specification can be embellished with many background specifications that will help us to understand risks associated with one or more elements of the requirement specification [12].

Background information must not be scattered around in different documents and meeting notes. It needs to be directly integrated into a sole master reusable requirement specification object. Otherwise it will not be available when it is needed: it will not be updated, or shown to be inconsistent with emerging improvements in the requirement specification.

See Figure 13 for a requirement template for function specification [5, page 106], which hints at the richness possible for background information.

TEMPLATE FOR FUNCTION SPECIFICATION <with hints>

Tag: <Tag name for the function>.
Type: <{Function Specification, Function (Target) Requirement, Function Constraint}>.

Basic Information
Version: <Date or other version number>.
Status: <{Draft, SQC Exited, Approved, Rejected}>.
Quality Level: <Maximum remaining major defects/page, sample size, date>.
Owner: <Name the role/email/person responsible for changes and updates to this specification>.
Stakeholders: <Name any stakeholders with an interest in this specification>.
Gist: <Give a 5 to 20 word summary of the nature of this function>.
Description: <Give a detailed, unambiguous description of the function, or a tag reference to someplace where it is detailed. Remember to include definitions of any local terms>.

Relationships
Supra-functions: <List tag of function/mission, which this function is a part of. A hierarchy of tags, such as A.B.C, is even more illuminating. Note: an alternative way of expressing supra-function is to use Is Part Of>.
Sub-functions: <List the tags of any immediate sub-functions (that is, the next level down), of this function. Note: alternative ways of expressing sub-functions are Includes and Consists Of>.
Is Impacted By: <List the tags of any design ideas or Evo steps delivering, or capable of delivering, this function. The actual function is NOT modified by the design idea, but its presence in the system is, or can be, altered in some way. This is an Impact Estimation table relationship>.
Linked To: <List names or tags of any other system specifications, which this one is related to intimately, in addition to the above specified hierarchical function relations and IE-related links. Note: an alternative way is to express such a relationship is to use Supports or Is Supported By, as appropriate>.

Measurement
Test: <Refer to tags of any test plan or/and test cases, which deal with this function>.

Priority and Risk Management
Rationale: < Justify the existence of this function. Why is this function necessary? >.
Value: <Name [Stakeholder, time, place, event>]: <Quantify, or express in words, the value claimed as a result of delivering the requirement>.
Assumptions: <Specify, or refer to tags of any assumptions in connection with this function, which could cause problems if they were not true, or later became invalid>.
Dependencies: <Using text or tags, name anything, which is dependent on this function in any significant way, or which this function itself, is dependent on in any significant way>.
Risks: <List or refer to tags of anything, which could cause malfunction, delay, or negative impacts on plans, requirements and expected results>.
Priority: <Name, using tags, any system elements, which this function can clearly be done after or must clearly be done before. Give any relevant reasons>.
Issues: <State any known issues>.

Specific Budgets
Financial Budget: <Refer to the allocated money for planning and implementation (which includes test) of this function>.

Figure 13 A template for function specification [5, page 106]

Principle 8. Carry out specification quality control (SQC)

There is far too little quality control of requirements against relevant standards. All requirements specifications ought to pass their quality control checks before they are released for use by the next processes. Initial quality control of requirements specification, where there has been no previous use of specification quality control (SQC) (also known as Inspection), using three simple quality-checking rules ('unambiguous to readers', 'testable' and 'no optional designs present'), typically identifies 80 to 200+ words per 300 words of requirement text as ambiguous or unclear to intended readers! [13]

Principle 9. Consider the total lifecycle and apply systems-thinking - not just a focus on software

If we don't consider the total lifecycle of the system, we risk failing to think about all the things that are necessary prerequisites to actually delivering full value to real stakeholders on time. For example, if we want better maintainability then it has to be designed into the system. If we are really engineering costs, then we need to think about the total operational costs over time. This is much more than just considering the programming aspects.

You must take into account the nature of the system: an exploratory web application doesn't need to same level of software engineering as a real-time banking system!

Principle 10. Recognise that requirements change: use feedback and update requirements as necessary

Ideally requirements must be developed based on on-going feedback from stakeholders, as to their real value. System development methods, such as the agile methods, enable this to occur. Stakeholders can give feedback about their perception of value, based on the realities of actually using the system. The requirements must be evolved based on this realistic experience. The whole process is a 'Plan Do Study Act' Shewhart

cyclical learning process involving many complex factors, including factors from outside the system, such as politics, law, international differences, economics, and technology change.

Attempts to fix the requirements in advance of feedback, are typically wasted energy (unless the requirements are completely known upfront, which might be the case in a straightforward system rewrite with no system changes). Committing to fixed requirements specifications in contracts is not realistic.

Who or What Will Change Things?

Everybody talks about requirements, but few people seem to be making progress to enhance the quality of their requirements specifications and improve support for software engineering. Yes, there are internationally competitive businesses, like HP and Intel that have long since improved their practices because of their competitive nature and necessity [8, 14]. But they are very different from the majority of organizations building software. The vast majority of IT systems development teams we encounter are not highly motivated to learn or practice first class requirements (or anything else!). Neither the managers nor the systems developers seem strongly motivated to improve. The reason is that they get by with, and even get well paid for, failed projects.

The universities certainly do not train IT/computer science students well in requirements, and the business schools also certainly do not train managers about such matters [15]. The fashion now seems to be to learn oversimplified methods, and/or methods prescribed by some certification or standardization body. Perhaps insurance companies and lawmakers might demand better industry practices, but I fear that even that would be corrupted in practice if history is any guide (for example, think of CMMI and the various organization certified as being at Level 5).

Summary

Current requirements specification practice is often woefully inadequate for today's critical and complex systems. Yet we do know a considerable amount (Not all!) about good practice. The main question is whether your 'requirements' actually capture the true breadth of information that is needed to make a start on engineering value for your stakeholders.

Here are some specific questions for you to ask about your current IT project's requirements specification:

- Do you have a list of top-level critical objectives?
- Do you consider multiple stakeholder viewpoints?
- Do you know the expected stakeholder value to be delivered?
- Have you quantified your top five quality attributes? Are they are testable? What are the current levels for these quality attributes?
- Are there any optional designs in your requirements?
- Can you state the source of each of your requirements?
- What is the quality level of your requirements documentation? That is, the number of major defects remaining per page?
- When are you planning to deliver stakeholder value? To which stakeholders?

If you can't answer these questions with the 'right' answers, then you have work to do! And you might also better understand why your IT project is drifting from delivering its requirements. The good news is that the approach outlined in this article should allow you to focus rapidly on what really matters to your stakeholders: value delivery.

References

1. Thomas Carper, Report Card to the Senate Hearing "Off-Line and Off-Budget: The Dismal State of Federal Information Technology Planing", July 31, 2008. See http://uscpt.net/CPT_InTheNews.aspx [Last Accessed: August 2010].

2. The Standish Group, "Chaos Summary 2009", 2009. See http://www.standishgroup.com/newsroom/chaos_2009.php [Last Accessed: August 2010].

3. John McManus and Trevor Wood-Harper, "A Study in Project Failure", June 2008. See http://www.bcs.org/server.php?show=ConWebDoc.19584 [Last Accessed: August 2010].

4. David Yardley, Successful IT Project Delivery, Addison-Wesley, 2002. ISBN 0201756064.

5. Tom Gilb, "Competitive Engineering: A Handbook for Systems Engineering, Requirements Engineering, and Software Engineering using Planguage", Elsevier Butterworth-Heinemann, 2005.

6. Jennifer Stapleton (Editor), DSDM: Business Focused Development (2nd Edition), Addison Wesley, 2003. ISBN 0321112245. First edition published in 1997.

7. Mike Cohn, User Stories Applied: For Agile Software Development, Addison Wesley, 2004. ISBN 0321205685.

8. Trond Johansen and Tom Gilb, From Waterfall to Evolutionary Development (Evo): How we created faster, more user-friendly, more productive software products for a multi-national market, Proceedings of INCOSE, 2005. See http://www.gilb.com/tiki-download_file.php?fileId=32

9. Dr. Billy Vaughn Koen, "Discussion of the Method: Conducting the Engineer's Approach to Problem Solving", Oxford University Press, 2003.

10. Tom Gilb, Real Requirements, see http://www.gilb.com/tiki-download_file.php?fileId=28

11. Taiichi Ohno, "Toyota production system: beyond large-scale production", Productivity Press, 1988.

12. Tom Gilb, "Rich Requirement Specs: The use of Planguage to clarify requirements", see http://www.gilb.com/tiki-download_file.php?fileId=44

13. Tom Gilb, Agile Specification Quality Control, Testing Experience, March 2009. Download from www.testingexperience.com/testing experience01_08.pdf [Last Accessed: August 2010].

14. Top Level Objectives: A slide collection of case studies. See http://www.gilb.com/tiki-download_file.php?fileId=180

15. Kenneth Hopper and William Hopper, "The Puritan Gift", I. B. Taurus and Co. Ltd., 2007. ■

Getting the truth, the whole truth and nothing but the truth from your Test Management Tool… Dream or Reality?

basic

Author: ***Eric RIOU du COSQUER***

About the author:

Eric RIOU du COSQUER is responsible for a team dedicated to Requirements and Tests Management at Orange, a worldwide French telecommunication company. In his current position, he is in charge of defining and implementing the processes dedicated to Requirements and Test management and also to select and support the associated tools. His daily activities also consist in providing internal training courses and support to the software projects of the Information system division of France Télécom.
He owns the Foundation Level and TestManager/Test Analyst Advanced Level certificates of the International Software Testing Qualifications Board.
Contact : eric.riouducosquer@orange.fr

Abstract

Ten years ago it was a delicate business to convince people to use a Test Management Tool! Nowadays the situation seems to be far better since most of the IT projects are using that type of tool. But do you think the legitimate expectations of the Test Manager and Project Manager are fulfilled? According to my experience in supporting and auditing Test Projects for the Information System Division of a large Telecommunication company, I would unfortunately answer « No! » without any hesitation. The goal of this presentation is not only to make you understand why the test management tool too often turns into a gasworks but also to help you obtain the best from your tool.

1 Introduction

In order to get all the test-related information you need with complete confidence, let's go back to the basic principles and success criteria to be implemented!

2 Rethink the main reasons why you are using a Test Management Tool!

2.1 Three high level abstract reasons

2.1.1 Cost, Delay, Quality

The Return on Investment of the Test Management Tool is easy to demonstrate.
The main costs are the following:
- Expense 1 : software licenses, required hardware and maintenance costs (unless you are using an open source tool)
- Expense 2 : Trainings (cost of the training itself and cost of the time spent by the participants)
- Expense 3 : Time spent using the tool
- Gain 1 : Time saved compared to the same job carried out with Excel sheet or other documents
- Gain 2 : Positive Business Impact due to a higher quality and a smaller number of failures

2.1.2 Keep the customer satisfied

Keeping the customer satisfied is also a good reason to use a test management tool. The tool will directly contribute to a higher "technical" quality of the final product. Actually, having a good "technical" quality is not enough to satisfy the customer! One of the key points here is the requirements coverage that should ensure not only a good "technical" quality but also a good "functional" quality.

2.1.3 Offer a more interesting job to the testers!

Doing the same task as previously but with a specific tool may be quite

motivating if the tool is user friendly.
Unfortunately it is rarely the first reason!

2.2 Eight Low level concrete reasons… (More valuable)

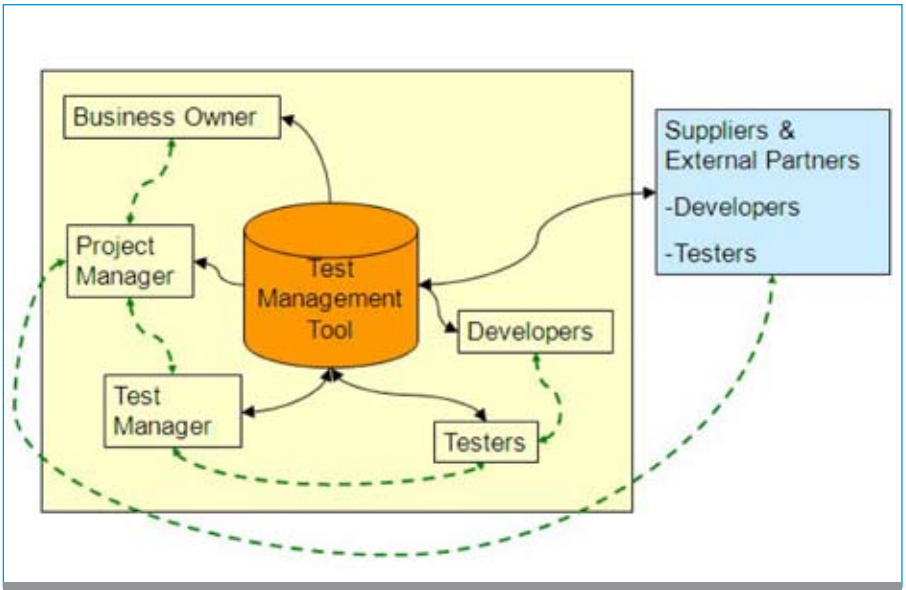
2.2.1 A good cooperation between the stakeholders

Business Owner, IT Project manager, Test Manager, Testers, Suppliers and other external partners use the Test Management tool in different ways.
A Business Owner is expecting Proofs of testing and a status of the Requirements implementation...

An IT Project manager has to get information from the bottom and provide information to the top.

A Test Manager has to manage the testing activities quickly, get and provide information to the Project manager and check outsourced testing activities.
Testers need to know what to do while tests execution.

Suppliers and Other External Partners have to build their tests from the right requirements and provide visibility on their testing activities.



Picture 1 A good cooperation between the stakeholders

2.2.2 Centralization (tests, tests executions and results)

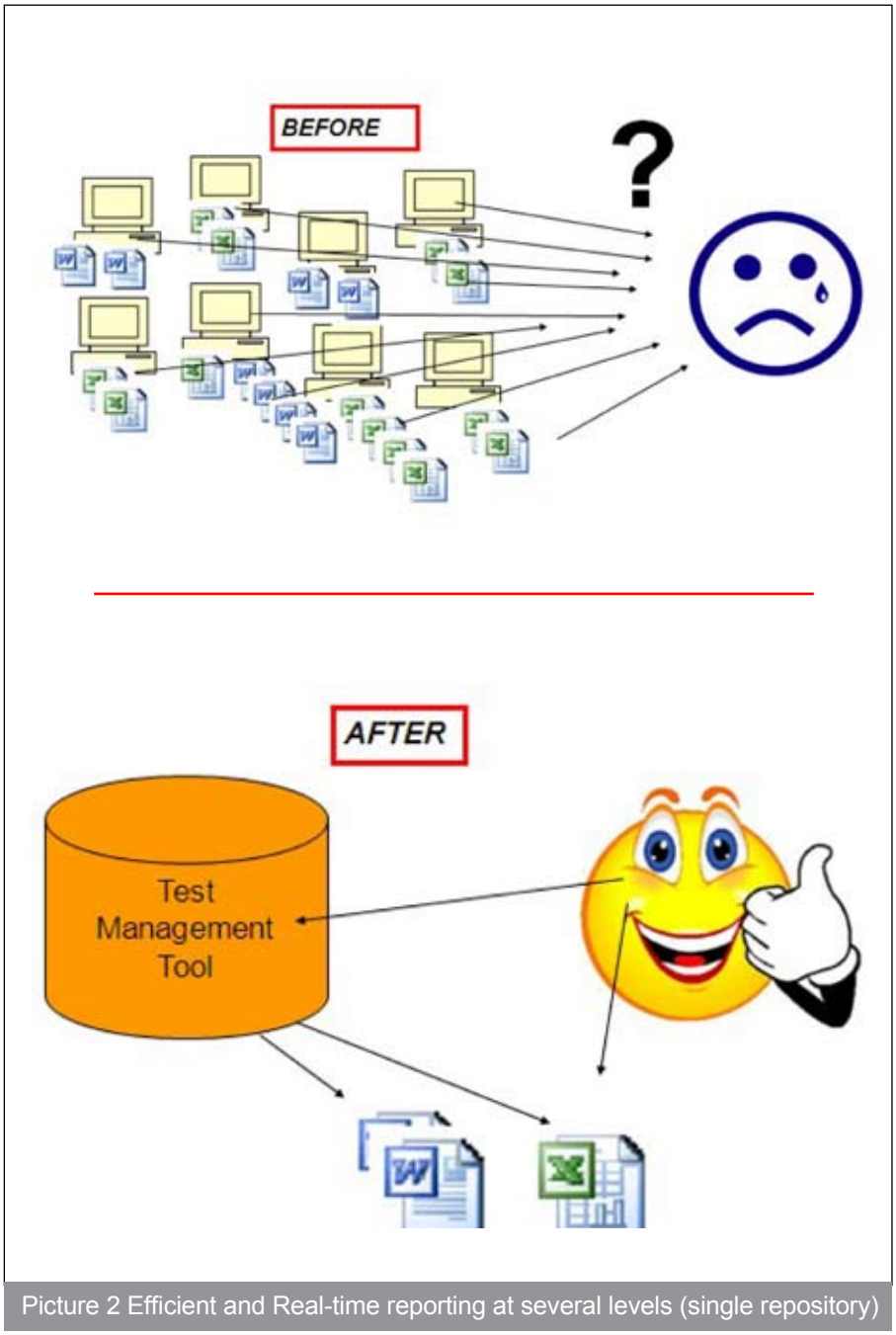
Everything is at the same place, no more files to look after, and no more versions to fight with. Isn't it a good reason?

2.2.3 Easier reporting at several levels (Document generation)

Depending on your needs and your position, the tool should be able to easily generate the reports you need to make a decision or give an approval.
Testers need to know what they still have to do.

Tests manager must be able to quickly answer the following questions: What are the remaining testing tasks? What is the real status of the tested components, applications or system?

A project manager needs the answer to the following questions: How is testing going on? Where are the main weaknesses in the tested items? What is going wrong in



Picture 2 Efficient and Real-time reporting at several levels (single repository)

my development process?

2.2.4 Real time follow-up

Everything is up to date and available in a test management tool.

No need to look after documents or people to get the information!

2.2.5 A record of all the results

All the results (even discussions, changes and comments) are recorded with detailed information: date, time, author…

2.2.6 Facilities to check the Requirements Coverage thanks to the Requirements-Tests link

Each test is testing something.
This « something » should be specified « somewhere ».

This « somewhere » should either be referenced as a Requirement or be the Requirement itself. If there is no specification regarding what is tested, a new requirement should be created.

2.2.7 Facilities to accelerate the Defects correction thanks to the link between Defects and Test execution results

You will save time by creating the defect during test execution.
The link between defects and tests execution result will help reproduce a defect by re-playing the associated test.
The link will also be useful to check the correction of a defect.

2.2.8 Reuse test artifacts

Thanks to the Test Management Tool, you will easily reuse test artifacts at several levels in a single project:
- From one version of an application to another
- Between projects of the same type
- To achieve regression testing

3 « Prerequisites »: What should you have before using the tool?

Before packing you should know where you are going…
Before using a test management tool you should know which elements will guide its implementation and the construction of its internal structure…

3.1 The Testing Process

It is one of those IT processes which is strongly related to the development process.

The testing process involves several kinds of roles and activities with expected deliverables.

The Test Management Tool is intended to optimize and facilitate the instantiation of the testing process.

Its internal organization is fully based on IT Processes, particularly on the testing process, and on their specific terms (vocabulary, activities, roles…)

3.2 Documents

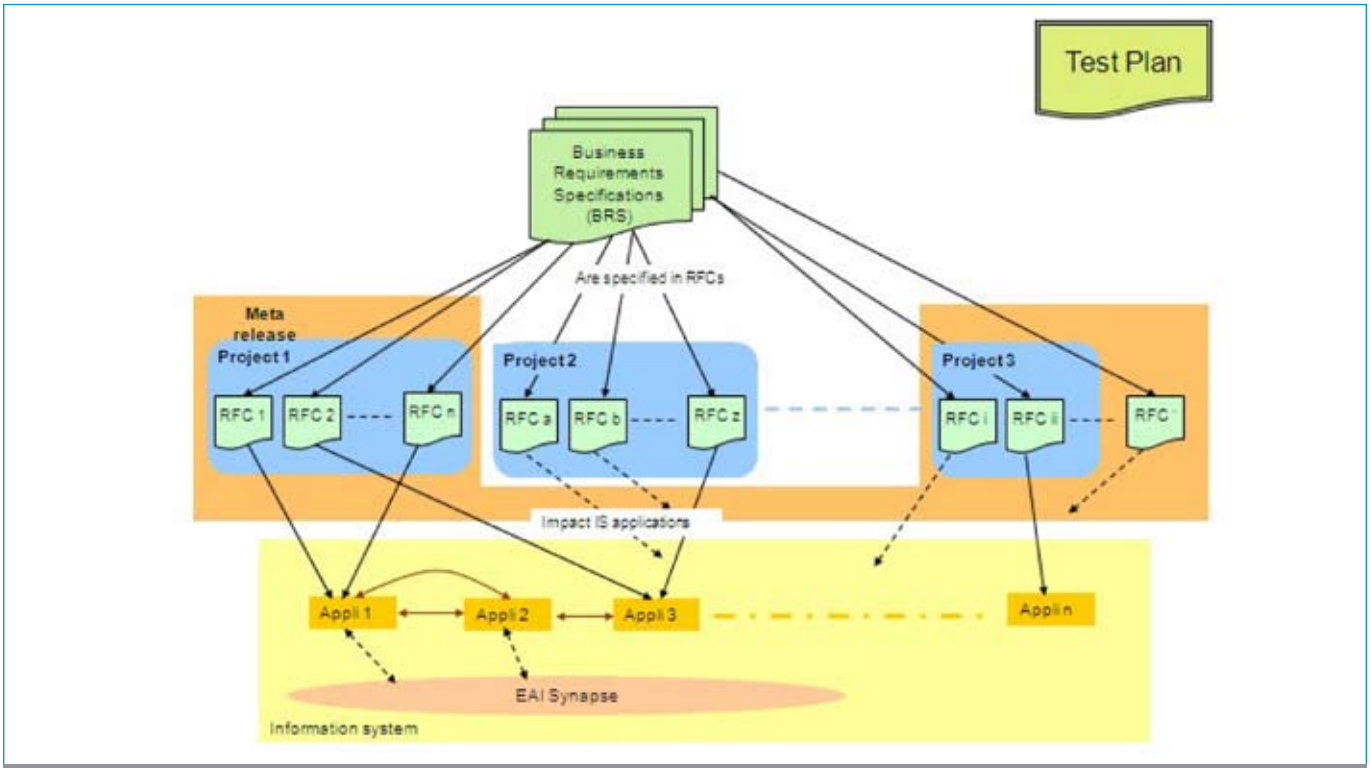
Each test is testing something. This « something » should be described somewhere.
Several documents include information that will be useful to create Requirements and Tests. Identify them!

3.3 Testing Levels in your context

Name and describe them!

3.4 Test Environments

Each test execution is carried out in a specific environment and its results often depend on the test environment. A single test may have different result if the environment is not the same.



Picture 3 Prerequisites: Delivery Concepts

This concept of Environment should be taken into account in the dynamic part of the tool (Test Suites & Test Execution).

4 How to motivate people (or diplomatically force them) to use the tool?

Get a high level Management Decision… eventually based on a local success.
Involve motivated people in a visible pilot project
Communicate and brainstorm from the beginning of the pilot phase.
Listen to people's expectations.
Clearly show how the tool will help.
Define, spray and control smoothly the rules.
Continuously help and support stakeholders, don't leave them alone once the test project is started!
Accept valuable compromises.
Organize a close cooperation between stakeholders using the tool.
Set up a regular reporting to all the pairs of a specific level and their managers.

5 How can we deal with synchronization and traceability issues?

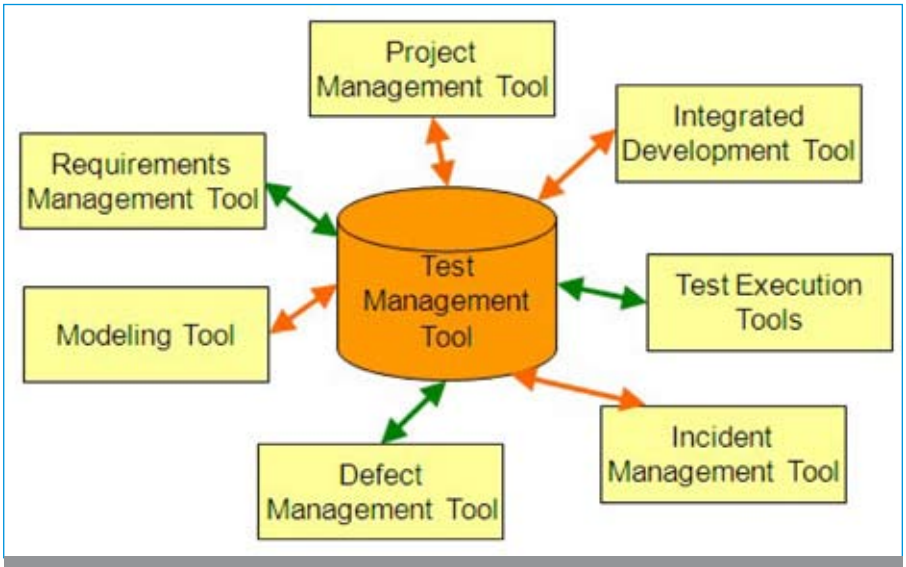
Requirements management tool, Defects management tool, Tests execution tools, Incident management tool, Configuration management tool, Modeling tool and Development tool are more or less associated to the Test Management Tool.
No vendor provides a suite that would integrate the functionalities of all those tools and it is not realistic to look for a perfect technical integration between all those tools.
The most important thing is to study carefully all the possible interactions and associated needs. Then you have to specify how to address each need, manually or with a specific development.

6 What can we do with the test items from one version to another?

A test may be modified and consequently may exist in several versions. The tested items may also exist in several versions. We may have version management at several levels. Depending on the tested version of a test item and on the version of the test, the results may be different. The configuration of the test environment may also be version managed
The consequence is that for each test execution, you must be able to identify the version of the tested item, the test and the test environment.

7 What is the right level of granularity for your rules and recommendations? And how can we make sure they are applied?

What for? You need rules and recommendations to ensure a proper use of the tool by all the stakeholders.
For what? You should provide rules for all the basic features of the tool.
Which level of granularity? It depends on the perimeter the tool is used. The largest it is, the lowest the level of granularity.
How to make sure they are applied? Check, check, check!



Picture 4 Cooperation with other tools

8 What kinds of training should be implemented?

You need 3 types of trainings:
- Basic use of the tool (independent from you context)
- Advanced use of the tool (independent from your context)
- Use of the tool in YOUR specific CONTEXT in an EFFICIENT way (context specific)

8.1 Basic use of the tool

In this first type of training, you should specify how to:
- Get an access to the tool!
- Create a Requirement
- Create a Test Case
- Associate Requirements and Test Cases
- Organize Test Cases into Test Suites
- Launch Test Suites
- Create Defects
- Associate Defects to Test Execution Results
- Follow-up the whole testing progress
- Use pre-defined Reports

8.2 Advanced use of the tool

In this second type of training, you should specify how to:
- Manage users and associated rights
- Customize project items (attributes describing a test...)
- Customize workflows
- Create script implementing specific needs
- Implement mail notification
- Define reports

8.3 Context-specific training

In this third type of training, you should:
- Describe the stakeholders involved in testing, respective roles and responsibilities
- Reference the main documents (Test Plan…)
- Describe the « structural » items
- Specify Rules to be applied while creating the structures (folder trees)
- Identify and describe the attributes needed for each item (Requirement, Test Case, and Defect)
- Describe procedures for the main actions
- Explain the recommended reporting (Which reports are available? How to generate them?) ■

What a Tester Should Know, At Any Time, Even After Midnight

intermediate

Author: **Hans Schaefer**

About the author:

Hans Schaefer
Software Test Consulting
Chairman, Norwegian Software Testing Qualifications Board
N-5281 Valestrandsfossen, NORWAY
www.softwaretesting.no
hans.schaefer@ieee.org

You can do the necessary testing «just as a job». This is neither good enough nor is it motivating for the tester. Alternatively, you can invest «the little extra», i.e. do a better job. In this article, I try to define what this «little extra» means.

Major facts are the idea of a tester as a "devil's advocate" in the chase for potential and real defects; the need to prioritize by risk and profit or importance; the use of facts about defects, such as their uneven distribution. But in order to do a good job, the tester must require quality of the product to be tested. There is the Tester's Bill of Rights. Finally there is the need to continuously learn: About defects, the application area, software development, and test methods and tools. This way, testing turns into an interesting and rewarding job, and the tester contributes effectively and efficiently to the project.

1. Testing the Normal Way is not Enough

Systematic testing of software or systems can be learned, just like any engineering discipline. There are tester knowledge certification schemes (ISTQB, ISEB, GTB), there are books (Myers 79, Beizer 95, Kaner 99, Copeland 2004,) and there are standards (ISTQB Glossary, BS 7925, IEEE 829, 1008, 1012,). At least the books and most of the standards have been around for a long time, and many techniques are widely accepted. This means testing can actually be studied and then executed in some systematic way. This does not mean that the typical testing methods described in this material are widely practiced (Schaefer 2004). But it is at least possible to do testing «by following the book».

For a tester, or test engineer, there are two major activities: Designing test cases, and executing test cases and observing and analyzing the results. If the results are not what was expected, deviations must be reported and followed up. Additionally, modern methods, such as exploratory testing (Bach website), include tasks like automation, and management of testing time in the tester's task list.

The normal way of performing a tester's job is to learn some techniques, follow these techniques, execute the test, and conclude the work with a test report. The typical task description tells people to «test the system», wihtout defining any more details. Some books define the task as «getting information about the quality», or «measuring quality» (Hetzel). As test execution is one of the last activities in a project, it is very often run under severe budget and time pressure. This means that the testers have a hidden or open motivation to compromise on the thoroughness of their job. For example, since the detection of defects will always delay both testing and the delivery or acceptance of the product, there is a hidden pressure not to find any defects, or at least not serious ones. Testing is just «done». This job is then not very motivating, investment in learning is scarce, people try to find a different job, and testing does not improve over time.

2. Can We Do Testing in a Better Way?

Glenford Myers, in 1979, defined testing differently: The aim of testing is to find defects. He used this definition because it motivates people. Having a negative focus, trying to break the product, leads to finding more and more serious defects than just checking whether the product «works».

Most people do as they are told. If they are told to find as many defects as possible, they will try to do so. If they are told to get the job done (and explicitly or inherently get the message that defects delay progress), people will try not to find

defects, or they will overlook many.

Thus the first rule is to clearly define the purpose of testing, and make the purpose perfectly clear to people. This will be discussed in section 3.

There is an additional problem with any job, not only testing: The world is developing, especially in software. New techniques, methods and tools become available or are used in design. Software products are more and more integrated with each other and growing more and more complex. The focus of the requirements is changing, for example towards emphasizing more security, interoperability and usability. This leads to changes in the requirements on the testing job. Thus, a tester should continuously try to learn more. This will be discussed in section 4.

The next problem is the mindset of people. Some people readily accept information they see or rules that are given. Other people are critical and investigate and ask. As one of the purposes of testing is to find problems and defects, a mindset that does not accept everything without asking, checking more details, getting more information or thinking would lead to better testing. This will be discussed in section 5.

Part of the tester's task is to report incidents. This is not easy. Most literature read by testers just describes issue and defect management, i.e. the life cycle of reporting, prioritizing and handling these. But this is just «the ordinary job». Actually, there is more to it. It can be compared to the task a frustrated user has when calling the supplier help desk: Describing the problem in such a way that the other side accepts it as important enough to do something about it. It means collecting more information about the trouble, but also to think about how to «sell» the bug report to the responsible person. This is the topic of section 6.

Finally, a tester has some rights. We should not just test anything thrown at us over the wall. If information we need is not available, or if the product to be tested is far too bad, testing it anyway would mean to waste resources. There should be some entry criteria to testing, some «Tester's Bill of Rights» (Gilb 2003). This is discussed further in section 7.

All of this has to do with the philosophy of testing. But there are some tools, some very basic rules for doing the work. There is a lot of controversy about what the basis is, but I dare to include a few from a conference speech (Schaefer 2004). This is topic of section 8.

There is definitely more to it. A tester should always be on the outlook for more challenges in field of testing. This paper is only the beginning of what you may look for.

3. The Purpose of Testing

There are a lot of possible goals for testing. One main, though possibly boring, purpose is to measure the quality of a product. Testing is then considered just a usual measuring device. Just usual measuring is not much fun, but is a necessary job, which must be done well. However, there are questions a tester should ask in order to measure optimally. The main question is which quality characteristics are most important to know, and how precisely the measurement must be performed.

Another definition of testing is trying to create confidence in the product. Confidence, however, is best built by trying to destroy it (and not succeeding in doing so). It is like scientific work: Someone proposes a new theory, and all the educated specialists in the area try all they can to show that it is wrong. After this has been tried unsuccessfully for some time, the new theory is slowly adopted. (Anyway, a theory is only valuable if it is concrete enough to offer the possibility to be fasified). This view supports Myers' definition of software testing: Find defects! The approach is a pessimist's approach. The pessimist believes «this probably does not work» and tries to show it. Every defect found is then a success.

People function by motivation. The definition of testing as actively searching for bugs is motivating, and people find more bugs this way. It works in two ways: One is by designing more destructive test cases or just simply more test cases. The other is by having a closer look at the results, analyzing details a non-motivated tester would overlook. In the latter case this may mean to analyze results that are not directly visible at the screen, but are deep inside some files, databases, buffers or at other places in the network.

A tester should try to find defects! Defects may appear in places where you do not see them easily, i.e. not on the screen output!

But it is more than this! "Defects are like social creatures: they tend to clump together." (Dorothy Graham, private communication). It is like mosquitoes: If you see and kill one, do you think this is the last one in the area? Thus we may have a deeper look in areas where we find defects. Testers should have an idea where to look more. They should study quality indicators, and reports about them. Indicators may be the actual defect distribution, lack of internal project communication, complexity, changes, new technology, new tools, new programming languages, new or changed people in the project, people conflicts etc. The trouble is that all these indicators may sometimes point in the wrong direction. The defects found may have been detected at nearly clean places, just because the distribution of test cases has tested these areas especially well. Project communication may look awful; some people who should communicate may be far from each other. But such people might communicate well anyway, in informal or unknown ways, or the interface between the parts they are responsible for may have been defined especially well, nearly "idiot-proof". Complex areas may be full of errors. There is a lot of research showing that different complexity indicators may predict defect distribution. However, there are nearly always anomalies. For example, the most experienced people may work with the most complex parts. Changes introduce defects or may be a symptom for areas, which have not been well analyzed and understood. But changes may also have been especially well thought out and inspected. In some projects, there are «dead dogs» in central areas that nobody wants to wake. These central areas are then badly specified, badly understood and may be completely rotten. But since nobody wants to «wake the sleeping dog» there are no change requests. New technology is a risk, partly because technology itself may lead to new threats, partly because using it is new to the people. People do not know what its challenges are. The same is true about the testers. Little may be known about the boundaries of technology and tools. However, it may work the opposite way: New technology may relieve us of a lot of possible defects, or simply make them impossible.

Finally we may look at the people involved. It is the people who introduce the defects. Research has shown that «good» and «bad» programmers have widely differing productivities and defect rates. However, defects do not only result from programming. It is more difficult to map people to design and specification trouble. But at least one factor nearly always has a negative impact: Turnover. If people take over other people's job, they very often do not get the whole picture, because tacit knowledge is not transferred. This is especially true if there was no overlap between people.
Thus, there are lots of indicators that may lead us to more defects, but we have to use them with care.

Defects clump together: they are social!
Defects often have a common cause!
Try to find common causes, and then follow them!
Where you find defects, dig deeper!

Another definition of testing is «measuring risk». This simply means that testing is a kind of risk mitigation, part of risk management. Testers should have an idea about product risk, as well as risk management. In the worst case, testers should ask questions about product risk, especially if nobody else asks these questions.

A very basic method for this is looking at the usage profile, and at possible damage. A tester should ask which kind of user would do what and how often. How will different users use the system? How will this vary over time? And a very important aspect is not to forget about wrong inputs and misuse. People have finger trouble, and interfacing systems have bugs. Thus there is not only use with correct inputs, but probably also a lot of use with wrong inputs. There are three kinds of tests: The good, the bad and the ugly tests: Good means everything is fine, all inputs are right. Bad means inputs are wrong. The ugly tests is where all hell breaks loose: Someone restarts the server while you do your reservation and at the same time sets the machine date wrong, …
The other risk factor is possible damage. This may be difficult to analyze. A start is to at least ask oneself: "What is the worst thing that can happen if this function, feature or request fails?"

Testing is risk mitigation.
What determines risk?
What happens if some input is wrong?

As a summary, it is best if the tester is a pessimist. (A pessimist is an optimist with experience). If something does not work, it is good news, because nobody will have the defect later. The positive effect will be felt in the long run. Better test forces developers to do better work, it informs management about risks, and it leads to lower cost (for repair). Testers bring bad news, but this is their job. Nobody loves speed checks on the motorway! But speed checks make our roads safer, and we all benefit.

A pessimist is a better tester!

4. Continuous Learning

Continuous learning is required in nearly every job. But for testers it is absolutely essential. In most cases, testing is done somehow systematically, using some black box approach. In addition, test design may follow heuristics. Any black box approach may leave important areas uncovered. Any heuristic approach is incomplete, as it is dependent on personal experience (or on learnt experience from others). And white box testing does not uncover errors of omission. It all comes down to this: If there is some aspect the tester doesn't know, it is not tested. Thus the tester should know as much as possible. But how?

A tester needs programming experience. There are lots of programming bugs, even after unit testing by programmers. (Unit testing is often not done well anyway). The tester should have an idea of what is difficult with the particular programming language used. As an example, loops and their counters are difficult in most cases, resulting in off-by-one errors. If the tester has no idea about these, s/he will not check loops with zero, one, maximum and more than maximum iterations, or will not check which individual object is selected. Then off-by-one errors will only be found by chance.

The tester needs design experience. Much design is about contracts and communication: Which module within which constraints and with which reactions should do which tasks? And where are these modules? How do they communicate and solve conflicts? If the tester has no idea about architectures and their inherent problems, s/he will have trouble planning (integration) tests.

The tester needs domain experience. System testing is about implementing the right solution, doing what the domain requires. Can you test a railway interlocking system? (Eh, what does interlocking mean, by the way...?). At least SOME domain experience should be there, and/or communication with different stakeholders.

The trouble is: This is not enough. Much about testing is getting the right information from the right people. Interview techniques are an interesting topic to learn. You get more information if you know how to interview people!

New systems interface with other systems in more and more intricate ways. And there are more and more unexpected interfaces. As an example, someone may integrate YOUR web service into HIS web site, together with other web services. Or your service works in a completely different way than someone else's, and is thus not attractive any more (or much more attractive than expected). This means: Testing for today's stakeholders may definitely not be enough. There are totally new ways your system may be used or interfaced, totally new ways in which it may be viewed, and you should try to anticipate at least part of this!

A tester should always try to find new ways of looking at the object under test, new approaches, and new viewpoints.

And finally: We want testers to use the newest approaches and technology. You have to learn them. Read testing books, look for and learn tools, study journals, participate in discussion groups, special interest groups, discuss with your colleagues, and go to testing conferences!

Learn more, about everything!
Programming, architecture, new domains, users, tools, anything!

«I am using three things to pull my equipment: dogs, dogs and dogs.» (Roald Amundsen)
For a tester, the three things are: Learning, learning and learning.

5. A Critical Mindset

Don't believe anything! A colleague of mine said: «Believing, that is the activity we do on Sunday morning in church. Everything else we should check.»

The trouble is: Believing is easier. It does not need any work. We just believe, because something is like expected, or because something is easier. Think about what is written in your newspaper. Is it really true? Where were the weapons of mass destruction? Was it really the Jews who were responsible for all this bad stuff? Is watching TV really dangerous for your kids? Is a certain soda really good for you?

The answer is: It is easiest not to ask. But if you question everything, you never get anywhere. Thus in our daily lives, we are accustomed to not ask and to take a lot of things for granted. To believe, or to assume, and TO NOT ASK QUESTIONS!

As a tester, don't assume anything. It may be wrong! Designers, specifiers, and programmers assume a lot. It may be difficult to ask because you may look stupid asking. The other part that could answer may be far away or not easily available. You don't even think there may be another interpretation. Or the other part doesn't know, or you get some sarcastic answer...

Using the pessimist view, you may as well assume that any assumption is probably wrong.

Don't assume! Ask!

There are ways of overcoming the trouble that you may look stupid. Learn how to deal with people, learn how to interview, learn how to be self-confident. (With regard to learning, see the section above). Ask someone else. Read, review, sleep over it, and try to find different interpretations. You may need a lawyer's mindset.

If you don't get an answer, have it on your issues list. But don't just assume something! Don't take things for granted! And especially: Don't believe «the system is probably right». There has been at least one banking system paying wrong interest. Difficult to check, after all... There was a geographical information system sending you around half the world instead of the direct way. There are airline reservation systems not telling you all available flights. Many more examples exist.

If nobody else asks the right question, you might do so!
Think about new possibilities, unknown problems, and the stuff you learn.
Think «out of the box»!

6. Defects

Nobody loves the messenger with bad news!

As a tester, most of what you report is bad news. (In a few cases there may be no bad news to bring, because everything seems to work, but that is a very different story).
The bad news is the bugs, or «issues» to call them in a neutral way. Textbooks handle this area well. There is issue reporting, registration, handling, removal, retesting, regression testing. We know all this. But there is something extra to it, and that is not found in the books very much:

1 - An issue is only interesting for a tester if it is accepted as a defect and repaired.
2 - There are defects, which are the result of running many test cases in a row in some very special order.

The first problem is one of salesmanship and discipline. As a tester, one has a sales job. Nobody is interested in spending any money on repairing defects. They will only be repaired if they are important enough. Thus, as a tester you have to report an issue in such a way that the developer understands that it must be repaired. The damage must look severe, the probability of it occurring must look high, and the issue must be easy to repeat.

Thus the tester should not just write an issue on the issues list. The tester should think: Are there any nearby scenarios that would make this problem worse? Are there more stakeholders interested in this? Is this really the whole problem or is there more to it? It is again «thinking out of the box». But it may also mean to invent and run some more test cases. Cem Kaner has presented some excellent material on this cause (Kaner bugadvoc).

Finally, there are the human issues, about diplomacy, politeness etc. A tester should make sure not to hurt anyone personally when reporting an issue. Someone said, "Diplomacy is the art of asking someone to go to hell in such a way that he will enjoy commencing the journey".

For every issue (or bug), research more about it!
Make sure you report it as a risk, and as the whole risk!
Defect reporting is a sales job!
Be diplomatic when reporting issues!

The second problem is worse: Sometimes we experience failures, and we cannot recreate them. For example, the first time the problem occurs, and the next time it is not there. These issues are called »intermittent bugs». They are especially difficult if they introduce system crashes. Upon restarting the system, any corrupted data in the memory may be deleted, destroying the evidence. In many cases, intermittent bugs are the result of long-term corruption of some resource or memory. One example are memory leaks. Some function in the program does not return unused memory after finishing. But because there is a lot of available memory, this can go on unnoticed for a long time, until the memory is depleted. It is even worse if this does not happen every time, but only in very special situations. But also other resources may be depleted. As an example, the Mars Explorer ceased working after 18 days due to too many files accumulated. (Luckily NASA could download new software). In many real-time embedded systems, the tasks are restarted at certain intervals in order to cancel out possible corruption of resources.

The trouble is that ANY shared resource can be corrupted. It comes down to checking the outputs which are not as easily visible as the screen: Files, memory pointers, buffers, databases, messages to remote systems, registry, anything. It could be anything. It could even be the result of a race condition, which depends on the exact timing of some parallel tasks. It is easy to see the screen output; everything else requires tools or extra actions from the tester. This may be too much work to do all the time. And intermittent bugs normally require a whole sequence of test cases, not just one input and output. Finally, it may be somewhere in the operating system, in the libraries, in something

that is not your fault.
However, if intermittent bugs occur, it is a good idea to be able to rerun the same sequence of test cases, maybe even with the same timing, and do more checking than before. James Bach (Bach 2005) has a good guide to investigating such problems:

Analyze even intermittent problems!
Log everything you do in testing!
Log everything you see, and look at more remote details!
Make it possible to rerun your tests, with more logging and analysis tools switched on!

One final problem: You may be wrong yourself. Humans err. Testers are humans. This means you overlook problems; you misunderstand outputs, and some of the problems you think you have found are actually no problem. Be self-critical: Sometimes it is your fault. You should also mistrust your memory. It is restricted. This means it is better to take notes, to log what you did, what the system did, what is installed etc. You can trust notes more than your memory.

You may be wrong – don't trust yourself 100%!
Take notes of what you do!

7. The Tester Has Some Rights

As a tester you have some rights.
Testing is often misused by others to clean up the mess. Instead of thinking beforehand, the defects are built into the system and the testers have to find them. This is a waste of both time and effort. A defect found by testers costs many times the effort, which would have prevented it, if it had been prevented. It also leads to extended time to deliver.

Testers should not be used to clean up, but to measure quality and report risk. It is plainly the wrong job.

A tester is NOT a vacuum cleaner!

The answer to the problem is using entry criteria. This means forcing the party before to do the job reasonably well. There are at least two sources where a tester's Bill of Rights has been published: Lisa Crispin talks about testers in Extreme Programming Projects.

The most important tester rights are these three:
* You have the right to make and update your own estimates (…).
* You have the right to the tools you need to do your job (…).
* You have the right to expect your project team, not just yourself, to be responsible for quality.

Tom Gilb (Gilb 2003) developed this list of testers rights (cited with the consent of the author):

Testers Bill of Rights:

1. Testers have the right to sample their process inputs, and reject poor quality work (no entry).
2. Testers have the right to unambiguous and clear requirements.
3. Testers have the right to test evolutionarily early as the system increments.
4. Testers have the right to integrate their test specifications into the other technical specifications.
5. Testers have the right to be a party to setting the quality levels they will test to.
6. Testers have the right to adequate resources to do their job professionally.
7. Testers have the right to an even workload, and to have a life.
8. Testers have the right to specify the consequences of products that they have not been allowed to test properly.
9. Testers have the right to review any specifications that might impact their work.
10. Testers have the right to focus on testing of agreed quality products, and to send poor work back to the source.

The last one is the real clue:

Testers should send bad work back to the source!

And: Testing is not the right answer. Prevention is!

Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with, and as a result the programming process will become cheaper. If you want more effective programmers, you will discover that they should not waste their time debugging; they should not introduce the bugs to start with!
Edsgar Dijkstra, "The Humble Programmer", ACM Turing Award Lecture 1972.

8. The Late Night Tester's Toolbox

How should a tester work? What should a tester always keep in mind when working?

One main trouble is to test "everything". This is far too much. It can never be achieved. But the tester should have some idea about what is tested and what not, or what is tested more or less. This relates to the **test coverage**.

In brief, there are three very basic concepts of coverage, and they can be applied to any notation, which is a diagram. For example a control flow diagram, a data flow diagram, a state transition diagram, a call graph, system architecture or a use case.
• Basic coverage executes every box.
• The next level of coverage is testing every connection.
• This should be the minimum for testing. If there is more time, the next level is combining things, for example pairs of connections.

A tester must be able to state what coverage a test has achieved!

Next, a test should follow the usage profile. This is difficult, especially in module and subsystem testing. But as a tester, one should at least try to get some idea about how the object under test will be used. If nothing can be said, the test should be directed at any use, testing robustness. This means that especially test cases for wrong inputs are of interest.

Follow the usage profile if possible!
If not this is not possible, test for robustness against wrong input.

One technique is the basis of most black box testing: Equivalence partitioning. It helps to save test effort, and it can be applied to derive other test techniques. As a tester, you should know it, but also be aware that it has caveats: Black box testing may leave out important aspects. You should also be aware that combination testing is of interest. Lee Copeland (Copeland 2004) has published a good introduction.

Equivalence partitioning is a good basic technique!
Remember combination testing!

Finally, there is all the test material. A worst-case scenario is if the tester has to admit that the test cannot be done or has been wrong. A big problem is the test environment, which should be prepared and tested early. Waiting for the test environment to work can kill any testing effort (and everybody else will point fingers!). After that, a defect may not be in the object under test, but in the test data or the output analysis. Be self-critical!

Test the test environment – well before test execution!
Check you test data!

And finally, there is test automation. A software product should be soft, i.e. easy to change. Change, however, is a risk. This means there is a need to test after any change. Retesting and regression testing may help. Running tests by using robots helps regression testing. But test automation is more than that: Tools may read specifications and automatically generate test cases. Tools may automatically create environments. Tools may be used to manage the testing effort and the test material.

Automate testing tasks!
Be aware that there is more automation than using test robots!

9. Selected References

• Bach 2005: James Bach. A blog note about possible causes of intermittent bugs: http://blackbox.cs.fit.edu/blog/james/

• Beizer 95: Boris Beizer, Black Box Testing, John Wiley, 1995

• Better Software Magazine, www.bettersoftware.com. www.stickyminds.com Very practical!

• BS7925: British Standard: www.testingstandards.co.uk/bs_7925-1.htm

• Copeland 2004: Lee Copeland, A Practitioner's Guide to Software Test Design, Artech House, 2004.

• Crispin: Lisa Crispin, Tip House, Testing Extreme Programming, Addison-Wesley, 2002, also http://home.att.net/~lisa.crispin/XPTesterBOR.htm

• Gilb 2003: "Testers Rights: What Test should demand from others, and why?"., Keynote at EuroSTAR 2003

• GTB: German Testing Board: www.german-testing-board.info The German Testing Board developed an earlier version of the current ISTQB certification.

• IEEE Standards: See www.ieee.org

• ISEB: Information Systems Examinations Board of British Computer Society. http://www.bcs.org/BCS/Products/Qualifications/ISEB/ has run a certification scheme for software testers since 1999.

• ISTQB: www.istqb.org International Software Testing Qualifications Board. Develops and runs an international software tester certification scheme.

• ISTQB Glossary: www.istqb.org/fileadmin/media/glossary-current.pdf

• Kaner 99: C. Kaner, J. Falk, H. Q. Nguyen, "Testing Computer Software (3rd ed.), John Wiley, 1999.

• Kaner bugadvoc: A presentation about how a tester should report issues. http://www.kaner.com/pdfs/BugAdvocacy.pdf

• Myers 79: Glenford Myers: The Art of Software Testing, John Wiley, 1979.

• Schaefer 2004; Hans Schaefer, "What Software People should have known about Software Testing 10 years ago - What they definitely should know today. Why they still don't know it and don't use it", EuroSTAR 2004

• About famous software errors: http://wired.com/news/technology/bugs/0,2924,69355,00.html?tw=wn_tophead_1

• About assumptions: "Never Assume", Sofie Nordhammen from St. Jude Medical at EuroSTAR 2005. ("Knowing What's Right, Doing What's Wrong") ■

Efficient Testing with All-Pairs

advanced

Author: **Bernie Berger**

About the author:

Acknowledgements: This presentation draws upon material from various sources, especially Lessons Learned in Software Testing by Kaner, Bach and Pettichord. Many thanks to them for their time and advice. Thanks to the AETG team who developed the all-pairs approach. Thanks to the reviewers of this paper for their insightful feedback: Lawrence Nuanez, Michael Steinhart, Dmitry Shchelokov

Introduction

If you're a software tester who's been in the field for a few years, you may have found yourself in one of the following situations:

• You're working as hard as you can to find bugs in a huge system and you can't get to everything within the deadline. You've already stumbled across some good bugs, and you think there are more in there, but the deadline comes and the software is released. A week later, a major client finds a serious problem with the new release and lets everyone know about it in an industry press release. You begin thinking about what went wrong and how you can improve your testing coverage.

• You're on a job interview, and the person across the desk asks you how to test a product, especially when there's too much to do in so little time.

• Your management has an elementary understanding of software testing, and as a result, sets unrealistic expectations for you to "test everything." They demand 100% coverage, including testing all possible inputs, from all possible interfaces, into all possible system paths, into all possible outputs. You know these are ridiculous demands, and you start thinking about alternate testing methods. (and/or alternate employment opportunities).

Well, I'm able to cite these examples because I've been in each of these situations myself. A few years ago, I started thinking about the Coverage question of testing software – how can you "test everything" without really testing everything? How can you test efficiently: to minimize testing efforts but maximie testing results?

I found a method that I enjoy so much that I use it and talk about it as often as possible. I've seen this technique referred to as "Pairwise Testing," "Combinatorial Method," and "Orthogonal Arrays" (actually, each of these is similar but different), but I'll use the term "All-Pairs."

In All-Pairs test design, we are concerned with variables of a system, and the possible values each variable could take. We will generate testcases by pairing values of different variables. Don't re-read that last sentence -- generating testcases using All-Pairs is easier than it sounds. It's like learning a new card game – at first you have to learn the object of the game, all the rules, and all the exceptions to the rules, and then the tips and strategies. But after you've played a few times, it seems naturally easy to play. It's the same here. The best way to explain how it works is with an example, so keep reading…

Cartesian Products

A good starting point for a discussion about all-pairs is with Cartesian Products. A Cartesian product is a scenario in which every unit of a group is matched with every unit of every other group, so that all combinations of units are achieved across all groups.

For example, imagine the following simple software application: A one-screen GUI, with two drop down lists and an 'OK' button. List1 contains three values, '0', '1', and '2'; List2 contains two values, '10', and '20'. The user selects one value from each list, and after clicking OK the product of the two values is displayed on the screen. The variables and their

List1	List2
0	10
1	20
2	

values look like this:

How would you test this program? How many input combinations are there? How long will it take to test all input combinations?

There are 3x2= 6 possible combinations. The 6 resulting combinations are a result of the Cartesian product of the two inputs,

List1	List2
0	10
0	20
1	10
1	20
2	10
2	20

and would look like this:

Each value of each variable is matched so that you have achieved all combinations. You can execute these tests manually in less than two minutes. Let's add a

little more complexity; Version 2.0 of our program has some new features:

List1 now contains integer values 0 through 9, and List2 was changed to an input Textbox, allowing all integers between 1 and 99. Additionally, there are some new checkboxes. When checked, checkbox1 multiplies the product by -1 (makes it negative). Checkbox2 will multiply the product by itself (gives the product's square). The variables and values now look like this [Figure 1]

List	Textbox	Checkbox1 (-x)	Checkbox2 (x^2)
0	1	On	On
1	2	Off	Off
2	3		
3	4		
4	…		
5	96		
6	97		
7	98		
8	99		
9			

Figure 1

Now I ask the same questions: How would you test this program? How many combinations are there? How long will it take to test all combinations?

Now there are 99x10x2x2=3,960 possible valid input combinations. There are also a host of invalid ones. (Note that the Textbox introduces a new concept – the possibility of invalid input. We will discuss that soon.) What to do?

Don't Use All Combinations

At the heart of all-pairs test design is the idea that you don't need to achieve all combination testing. Let's think about all combinations for a minute, in terms of finding bugs. In the previous example, let's say there is a bug where the program freezes when List=0 and Checkbox1 is on. That is, the program gets confused if it has to multiply zero by -1. That's not an unrealistic possibility, right? Now, if you were testing all combinations, how many test cases would fail because of this one bug?

99! That seems like a huge waste of effort, no? The fact is , the value in the Textbox is irrelevant to finding this bug. You could have found this bug with only one testcase – one that pairs a List=0 with a Checkbox=On – you don't need the extra 98 cases to find it.

What are you saying when you're testing all combinations? You're saying that you're looking for a bug that will only appear when one particular set of values across all your variables fail. Let me say that again in a different way: Each and every variable in the application has to be set to a particular value setting for the bug to appear. If even one value was changed, you wouldn't get that same bug. You're looking for a very specific set of conditions. Even in this nonsense calculator example, you have a 1:3,960 chance of failure. Will there be a bug that occurs once (and only once) out of 3,960 distinct input combinations? (And if by chance, there happens to be one, how low a priority do you think it would be? Picture the bug report…If there were 20 variables in your application, a bug description would look something like: "When amount is 3, and security level is set to high, and color is green, and filter is on, and day is Tuesday, and, and, and…<repeat 15 more times>… then the calculated output is incorrect")

In reality, all combinations is usually overkill testing. I should say though, that in mission-critical or safety-critical applications, all combinations might be a rational approach, such as pharmaceuticals or military defense systems, but that discussion is outside our scope.

Let's not be concerned with attempting to test combinations of all variables. Most bugs are found when only two variable values conflict, not when all conflict at the same time. In a recent NIST analysis of medical software device failures, only three of 109 failure reports indicated that more than two conditions were required to cause the failure (Wallace 2000). This is the main idea of all-pairs. It is more likely that you will find your bugs as a result of two values conflicting. It is far less likely that you need ALL variables in a particular value formation. Don't test all combinations. Test all-pairs.

OK, so that's the theory. How do you put it into practice? How do you figure out what pairs of variable values you have, and how do you fit them into actionable test scenarios?

Again, this is most easily explained with an example.

All-Pairs Working Example

First, figure out what your parameters (variables) are, and what each variable's possible values could be.

Organize this information in a spreadsheet.

Next, simplify your values. Group them. Use boundaries. For example, in the Textbox, instead of listing every valid value between 1 and 99, choose a smart representative sample. (As a rule of thumb, unless you have other specific information, use min-1, min, min+1, max-1, max, max+1.) [For more Information about this technique, see the material on Equivalence Class Partitioning in Testing Computer Software]

In our example, the user has more freedom to enter invalid choices in the Textbox than in the drop down list. Realize that when input is non-discrete, you can still group it into values. For example, value categories for freeform text might be: all alpha chars, all numeric chars, uppercase, lowercase, words in single quotes, keywords, etc.

In our example, to keep it simple for now, let's reduce the 99 valid values plus the infinite number of invalid values down to three: any valid integer, **any invalid integer, and any alpha chars** (which would be invalid).

Let's also reduce the 10 values in the dropdown list to two: 0, and any other value. Now our variables and values look like this [Figure 2]

List	Textbox	Checkbox1 (-x)	Checkbox2 (x^2)
0	Valid int	On	On
Any other	Invalid int	Off	Off
	Alpha		

Figure 2

Next, put your variables across the top header row in a table. Order your variables so that the one with the most number of values is first and the least is last. Here, we put Textbox first because it has 3 values. The other variables each have 2 values. [Figure 3]

Textbox (3)	List (2)	Negative (2)	Square (2)

Figure 3

Now we will start filling in the table. Each row of the table will represent a unique test case/scenario. We will fill the table column by column. Look at how many values there are in column 2. Here, we see that the List column has 2 values. That's how many times you will need to insert the values of the first column, Textbox. So we begin: [Figure 4]

Textbox (3)	List (2)	Negative (2)	Square (2)
Valid integer			
Valid integer			
Invalid Integer			
Invalid Integer			
Alpha			
Alpha			

Figure 4

We inserted six rows. The three values of Textbox, each repeated twice. Also notice that we skipped a row between each set of values. This is important – we will get to that soon.

Now, we fill in column 2. For each set of values in column 1, we will put both values of column 2 like so: [Figure 5]

Textbox (3)	List (2)	Negative (2)	Square (2)
Valid integer	0		
Valid integer	Other		
Invalid Integer	0		
Invalid Integer	Other		
Alpha	0		
Alpha	Other		

Figure 5

So far, so good. We have paired values across our two first variables. We can do a quick check… Valid and 0, Valid and Other. Invalid and 0, Invalid and Other. Alpha and 0, Alpha and Other. It's all good.

Let's continue on to the third variable. Column three is the checkbox that determines whether you want to multiply the product by -1. There are two values, on and off. Let's put in the ons and offs in column 3 and see what happens. [Figure 6]

Textbox (3)	List (2)	Negative (2)	Square (2)
Valid integer	0	On	
Valid integer	Other	Off	
Invalid Integer	0	On	
Invalid Integer	Other	Off	
Alpha	0	On	
Alpha	Other	Off	

Figure 6

Let's check to make sure we have all our pairs between column 3 and column 2. We have a 0 and On, but wait – there's no 0 and Off. We have an Other and Off, but there's no Other and On. Let's swap around the values in the second set in the third column: [Figure 7]

Textbox (3)	List (2)	Negative (2)	Square (2)
Valid integer	0	On	
Valid integer	Other	Off	
Invalid Integer	0	Off	
Invalid Integer	Other	On	
Alpha	0	*On*	
Alpha	Other	*Off*	

Figure 7

There. Much better. We have a 0/On, 0/Off, Other/On, and Other/Off. Notice that the last set on and off are arbitrary – we already have our pairs – and we don't care if the order is on/off or off/on.

Let's continue to the fourth column. This is the checkbox that multiplies the product by itself. There are also two settings, checked and unchecked. (I will call the values of this checkbox checked and unchecked so that they're different from the on and off in column 3, just for the example. You can have values of different variables called the same thing). We have to enter values in such a way that we get all our pairs. Let's give it a try: [Figure 8]

Textbox (3)	List (2)	Negative (2)	Square (2)
Valid integer	0	On	Checked
Valid integer	Other	Off	Unchecked
Invalid Integer	0	Off	Checked
Invalid Integer	Other	On	Unchecked
Alpha	0	*On*	Unchecked
Alpha	Other	*Off*	Checked

Figure 8

Once again, let's check our pairs. We have a 0/Checked and 0/Unchecked. We have Other/Checked and Other/Unchecked.

Good, now let's take a look at columns 3 and 4. We have On/Checked and On/Unchecked, and Off/Checked and Off/Unchecked. Not bad.

See, we fit every pair of values into six cases. If we were testing all combs, we would have 3x2x2x2=24. And if you consider that we reduced 99 to 3 in the Textbox, and 10 down to two in the dropdown list, that's an even bigger savings.

Remember when I said that skipping lines is important? Well, it is. Let's say Version 3.0 of our multiplier adds two more checkboxes. Checkbox3 will give the factorial value of the output, and Checkbox4 will convert the output into Hexadecimal notation. So we have to add two more columns to our table and enter our values. Let's continue with Checkbox3 in Column 5: [Figure 9]

Textbox (3)	List (2)	Negative (2)	Square (2)	Factorial (2)	Hex (2)
Valid integer	0	On	Checked	Yes	
Valid integer	Other	Off	Unchecked	No	
Invalid Integer	0	Off	Checked	No	
Invalid Integer	Other	On	Unchecked	Yes	
Alpha	0	On	Unchecked	No	
Alpha	Other	Off	Checked	Yes	

Figure 9

Let's make sure each column has at least one pair with our newly added fifth column. Looks like we did OK: Column 2 is OK: (0/Yes 0/No, Other/Yes, Other/No), Column 3 is OK: (On/Yes, On/No, Off/Yes, Off/No), and Column 4 is OK: (Checked/Yes, Checked/No, Unchecked/Yes, Unchecked/No). We're golden. Notice that the on off sequence in the last set in the third column is no longer arbitrary as it was when we had only three columns filled in. We need it in that order now for our new value pairs.

Textbox (3)	List (2)	Negative (2)	Square (2)	Factorial (2)	Hex (2)
Valid integer	0	On	Checked	Yes	Dec
Valid integer	Other	Off	Unchecked	No	Hex
Invalid Integer	0	Off	Checked	No	Hex
Invalid Integer	Other	On	Unchecked	Yes	Dec
Alpha	0	On	Unchecked	No	Dec
Alpha	Other	Off	Checked	Yes	Hex

Figure 10

Here we go one more time with the last column: [Figure 10]

Textbox (3)	List (2)	Negative (2)	Square (2)	Factorial (2)	Hex (2)
Valid integer	0	On	Checked	Yes	Dec
Valid integer	Other	Off	Unchecked	No	Hex
		On			**Hex**
Invalid Integer	0	Off	Checked	No	Hex
Invalid Integer	Other	On	Unchecked	Yes	Dec
		Off			**Dec**
Alpha	0	On	Unchecked	No	Dec
Alpha	Other	Off	Checked	Yes	Hex

Figure 11

And let's see how we did:

- Column 2 is OK. We have a 0/Dec 0/Hex Other/Dec Other/Hex.
- Column 3 is problematic: We do have an On/Dec and Off/Hex, but we're missing On/Hex and Off/Dec pairs.
- Column 4 is OK: Checked/Dec, Checked/Hex, Unchecked/Dec, Unchecked/Hex
- Column 5 is OK: We have a Yes/Dec, Yes/Hex, No/Dec, and No/Hex.

Textbox (3)	List (2)	Negative (2)	Square (2)	Factorial (2)	Hex (2)
Valid integer	0	On	Checked	Yes	Dec
Valid integer	Other	Off	Unchecked	No	Hex
Valid integer	*Other*	On	*Unchecked*	*No*	Hex
Invalid Integer	0	Off	Checked	No	Hex
Invalid Integer	Other	On	Unchecked	Yes	Dec
Invalid Integer	*Other*	Off	*Unchecked*	*Yes*	Dec
Alpha	0	On	Unchecked	No	Dec
Alpha	Other	Off	Checked	Yes	Hex

Figure 12

This time, we can't fit in our missing pairs (On/Hex and Off/Dec) by swapping around values. If we did, then other pairs would get out of whack. Instead, we simply add two more testcases that contain these pairs. Hence, the blank rows. [Figure 11]

The other variable values are purely arbitrary. They need to be filled in with some value, but we don't care which value, because we already have all our pairs. Go ahead and fill in the empty cells as you desire, and there you have it -- all-pairs in eight cases instead of all combinations in 96! [Figure 12]

I bet you're saying, this is really great, but do I have to go through this lengthy exercise with all this checking and swapping whenever I need an all-pairs analysis? This example is not nearly as complicated as the project I'm testing back home. It will take forever to figure it out with all my variables and values.

Is there a way to automate these calculations? If only there were a free downloadable script that calculated all-pair combinations…

Region[1]	Tier[2]	Property	Credit[3]	Residence[4]	LTV[5]	NIV[6]	NAV[7]	Refinance	CC[8]	Intro Rate[9]	Emp. D[10]
NY	L	1 fam	A+	Pri	80%	Yes	Yes	Yes	Cust	Yes	Yes
NJ	M	2 fam	A	Vac	90%	No	No	No	Bank	No	No
FL	H	3 fam	A-	Inv	100%						
TX	H+1	4 fam	B								
CA	H+2	Coop	<B								
DC	H+3	Condo									
Other											

Figure 13

Introducing James Bach's All-Pairs Tool

Fortunately for you, James Bach has already developed an all-pairs calculator. And because James is such a nice guy, it's free to download from his website at www.satisfice.com. He even gives you an instruction manual, a sample example, and the Perl source code. Let's take a quick look at it, and this time, let's use a real-life example.

You're testing an online mortgage application system. Using a web browser, users surf to the site and enter personal data into a series of forms. The system processes the data and, based on the data entered and the business logic programmed into the application, the system tells users what kind of mortgage products they qualify for and what their interest rate will be. Here's what the variable values look like: [Figure 13]

Description:

1 Location of property by US State
2 Amount borrowed.
3 Credit rating of applicant
4 Type of residence: Primary, Vacation, Investment
5 Loan to Value: amount of loan as percentage of value of property
6 No Income Verification
7 No Asset Verification
8 Closing Cost paid by Bank or Customer
9 Lower rate for first year of loan, followed by higher rate for subsequent years
10 Applicant is an employee of bank and gets a discount

There are 7x6x6x5x3x3x2x2x2x2x2x2 = 725,760 valid combinations. All-pairs does it in 50. Here's what to do:

1. Download and unzip ALLPAIRS from www.saticefice.com
2. Organize your variable and value matrix the way we did here and save it in tab-delimited text format. You can use Microsoft Excel and save as a .txt file. Remember to keep the formatting simple, and don't use spaces.
3. From a DOS command line, call the ALLPAIRS executable with the name of the .txt file as an argument. Redirect the output to an output file. For example,
ALLPAIRS.EXE MORTGAGE_IN.TXT > MORTGAGE_OUT.XLS
That's all there is to it.

Additional Considerations

Despite its ease of use, don't think that this tool is the silver bullet that will magically fix all your testing problems. You still need to think about how and when using all-pairs as a test technique is appropriate.

We already discussed briefly that it may not be an appropriate method for all situations. Here are some more points to consider:

Don't be tempted to find all-pairs of all variables, just because you can.

Sometimes, a particular variable will or will not exist, depending on the value of another variable. For example, let's say the same system that processes mortgages will also handle two other home finance products: home equity loans, and home equity lines of credit. There might be different business rules for processing line-of-credit applications. Maybe the different products are not offered in the same group of Regions, or perhaps there are different categories for LTV among the different home finance products.

It wouldn't make sense to add a "product" variable with values "Standard Mortgage", "Home Equity Loan" and "Line of Credit" into our all-pairs matrix because some of the resulting pairs (and therefore, testcases) would be undefined. If you did, you might generate a testcase that pairs a line-of-credit application with values that are not available for that product. This is different than negative testing, where you would be trying an option that the system doesn't expect. Here, you couldn't try that option at all because it doesn't exist. You would be creating buggy testcases.

One solution is to create separate all-pairs matrixes for the individual products. In this example, you can create individual all-pairs matrixes because the business rules are so different for each Home Equity product.

You won't find all your bugs by using this technique exclusively.

Remember, all-pairs only tests to see whether any two variables conflict, not if three or more conflict. Also, if you forget to include a variable, or you decide to exclude one from the matrix, its values won't be meshed with the rest of the variables. Lastly, reducing the number of values per variable (as we did with the Textbox in the first example) could cause

an important pair to be missed.

In Lessons Learned in Software Testing, Kaner, Bach, and Pettichord suggest to add additional cases, especially if you know of a specific combination that is widely used or likely to be troublesome.

Additionally, try to introduce the all-pairs technique to your current test process gently. If your current testing process isn't awful, then don't end it suddenly and replace it only with All-Pairs. All-Pairs is a great method to add to your testing toolbox.

Don't limit use of all-pairs to input variables

Throughout this presentation, we talked about variables as input to a system. Remember, variables can also mean test environments, paths through a system, and outputs. A common usage of all-pairs with non-input variables is in setting up test environments for Internet applications. Often, web apps need to be tested on a host of OS/Browser combinations. Nguyen illustrates this with an all-pairs example in Testing Applications on the Web.

Conclusion

Rooted in mathematical theory, the all-pairs technique is a thoughtful method when test planning. Download the all-pairs calculator and try it out. Using it, you can quickly generate test cases that have a good chance of finding bugs, instead of mindlessly copying and pasting text into testcase templates. The next time someone asks you to test "everything," or you're at an interview and you want to talk about test efficiency, remember this presentation and tell them about the benefits of the all-pairs approach.

References

1. Lessons Learned in Software Testing, Kaner, Bach, Pettichord

2. Testing Applications on the Web, Nguyen

3. Testing Computer Software, Kaner, Nguyen, Falk

4. "The Combinatorial Design Approach to Automatic Test Generation", Cohen, Dalal, Parelius, and Patton

5. "The AETG System: An Approach to Testing Based on Combinatorial Design" Cohen, Dalal, Fredman, Patton

6. "Orthogonally Speaking", Dustin, STQE Magazine 2001

7. "Orthogonal Array Testing Strategy (OATS) Technique", Harrell

8. "Converting System Failure Histories into Future Win Situations", Wallace and Kuhn

9. "ALLPAIRS", Bach

10. "A Dimensionality Model Approach to Testing and Imporving Software Robustness", Koopman, Pan, and Siewiorek ■

The Case for Peer Review

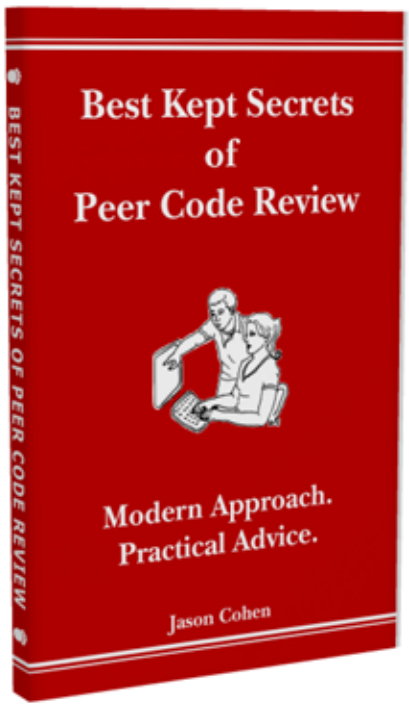
The $1 billion bug and why no one talks about peer code review.

Author: ***SmartBear Software***

basic

Reference:

This article is a single chapter of the book "Best Kept Secrets of Peer Code Review" by SmartBear Software. A compilation of 10 practical essays from industry experts, this book includes details of the largest case study on peer code review and gives specific techniques for effective reviews that your team can use right away.



SmartBear SOFTWARE

The full book can be accessed from the following link: http://smartbear.com/codecollab-code-review-book.php

It was only supposed to take an hour. The bad news was that we had a stack of customer complaints. The latest release had a nasty bug that slipped through QA. The good news was that some of those complaints included descriptions of the problem – an unexpected error dialog box – and one report had an attached log file. We just had to reproduce the problem using the log and add this case to the unit tests. Turn around a quick release from the stable branch and we're golden.

Of course that's not how it turned out. We followed the steps from the log and everything worked fine. QA couldn't reproduce the problem either. Then it turned out the error dialog was a red herring – the real error happened long before the dialog popped up, somewhere deep in the code.

A week later with two developers on the task we finally discovered the cause of the problem. Once we saw the code it was painfully obvious – a certain subroutine didn't check for invalid input. By the time we got the fix out we had twenty more complaints. One potential customer that was trialing the product was never heard from again.

All over a simple bug. Even a cursory glance over the source code would have prevented the wasted time and lost customers.

The worst part is that this isn't an isolated incident. It happens in all development shops. The good news? A policy of peer code review can stop these problems at the earliest stages, before they reach the customer, before it gets expensive.

The case for review: Find bugs early & often

One of our customers set out to test exactly how much money the company would have saved had they used peer review in a certain three-month, 10,000-line project with 10 developers. They tracked how many bugs were found by QA and customers in the subsequent six months. Then they went back and had another group of developers peer-review the code in question. Using metrics from previous releases of this project they knew the average cost of fixing a defect at each phase of development, so they were able to measure directly how much money they would have saved.

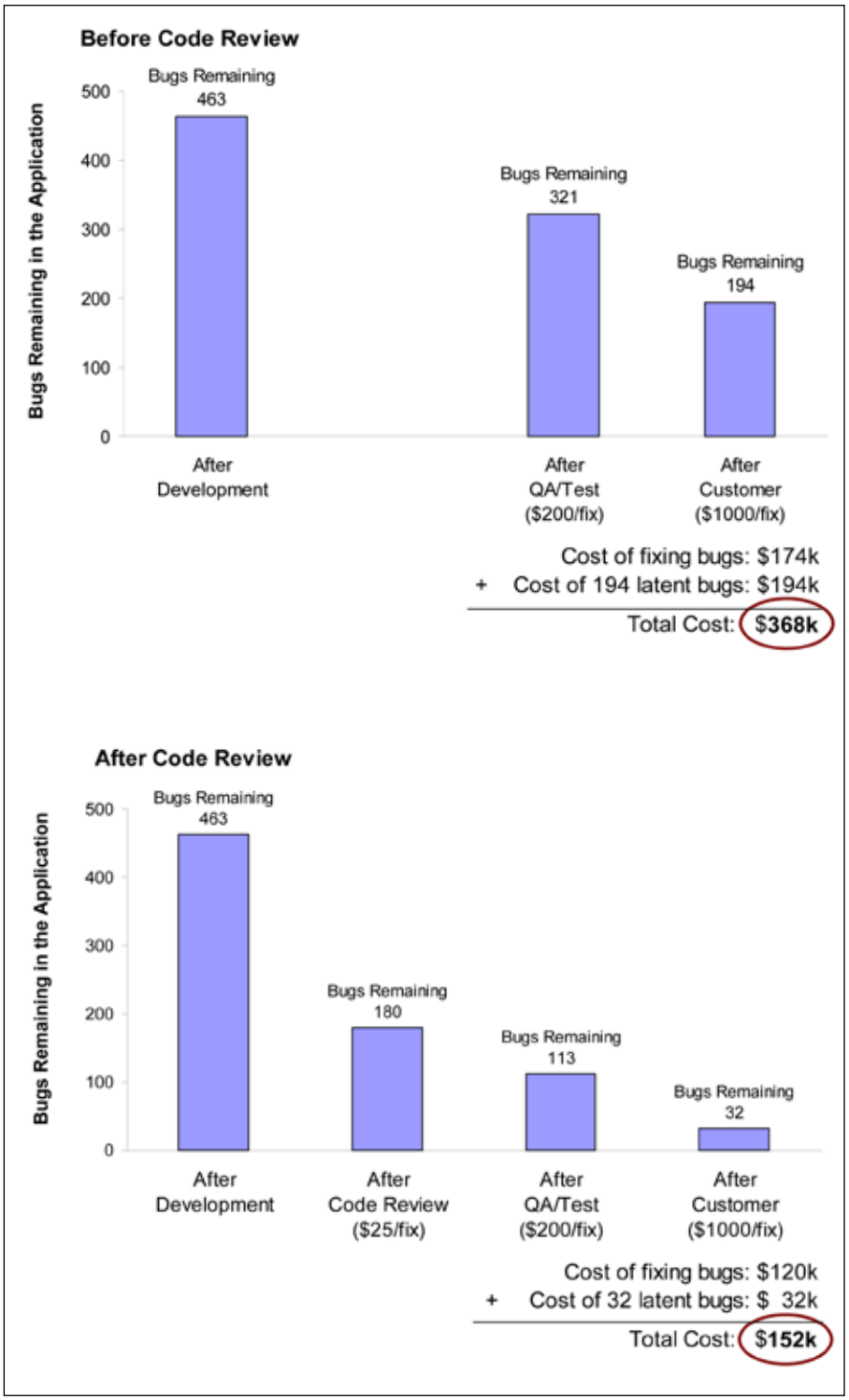
The result: Code review would have saved half the cost of fixing the bugs. Plus they would have found 162 additional bugs.

Why is the effect of code review so dramatic? A lack of collaboration in the development phase may be the culprit. With requirements and design you always have meetings. You bring in input from customers, managers, developers, and QA to synthesize a result. You do this because mistakes in requirements or architecture are expensive, possibly leading to lost sales. You debate the relative priorities, difficulty, and long-term merits of your choices.

Saving $150k: A real-world case study

Not so when actually writing the source code. Individual developers type away at the tasks assigned to them. Collaboration is limited to occasional whiteboard drawings and a few shared interfaces. No one is catching the obvious bugs; no one is making sure the documentation matches the code.

Peer code review adds back the collaborative element to this phase of



the software development process.

Consider this: Nothing is commercially published without corrections from several professional editors. Find the acknowledgments in any book and you'll find reviewers who helped "remove defects." No matter how smart or diligent the author, the review process is necessary to produce a high-quality work. (And even then, what author hasn't found five more errors after seeing the first edition?)

Why do we think it's any different in software development? Why should we expect our developers to write pages of detailed code (and prose) without mistakes?

We shouldn't. If review works with novels and software design it can also work when writing code. Peer code review adds a much-needed collaborative element to the development phase of the software development process.

The $1 billion bug

In 2005, Adobe attributed $1 billion in revenue to the PDF format[1].

Why is PDF worth $1 billion? Because it's the one format that everyone can view and print[2]. It just works. If it loses that status, Adobe loses the edifice built on that format, to which the fiscal year 2005 income statement attributes $1 billion.

Now imagine you are a development manager for Acrobat Reader, Windows Edition. The next major release is due in 9 months and you are responsible for adding five new features. You know how much is riding on Reader and how much revenue – and jobs – depends on its continued success.

So now the question: Which of those five features is so compelling, it would be worth introducing a crash-bug in Reader just to have that feature?

Answer: None!

Nothing is worth losing your position in the industry. But you still must implement new features to keep the technology fresh and competition at bay. So what techniques will you employ in your development process to ensure that no bugs get introduced?

Answer: Everything. Including code review.

Only code review will ensure that this code base – already over ten years old – remains maintainable for the next ten. Only code review will ensure that new hires don't make mistakes that veterans would avoid. And every defect found in code review is another bug that might have gotten through QA and into the hands of a customer.

There are many organizations in this position: The cost of losing market position is unthinkably large, so the cost of every defect is similarly large. In fact, any software company with a mature product offering is almost certainly in this position.

This doesn't mean they implement code review no matter what the costs; developer time is still an expensive commodity. It does mean that they're taking the time to understand this process which, if implemented properly, is a proven method for significantly reducing the number of delivered bugs, keeping

code maintainable, and getting new hires productive quickly and safely.

But you don't need to have $1 billion at stake to be interested in code quality and maintainability. Delivering bugs to QA costs money; delivering bugs to customers costs a lot of money and loss of goodwill.

But if code review works this well, why don't more people talk about it? Is anyone really doing it?

Why code review is a secret

In 1991, OOP was the Next Big Thing. But strangely, at OOPSLA there were precious few papers, light on content, and yet the attendees admitted to each other in hallway talk that their companies were fervently using the new techniques and gaining significant improvements in code reusability and in breaking down complex systems.

So why weren't they talking publicly? Because the development groups that truly understood the techniques of OOP had a competitive advantage. OOP was new and everyone was learning empirically what worked and what didn't; why give up that hard-earned knowledge to your competitors?

A successfully-implemented code review process is a competitive advantage. No one wants to give away the secret of how to release fewer defects efficiently.

When we got started no one was talking about code review in the press, so we didn't think many people were doing it. But our experience has made it clear that peer code review is widespread at companies who are serious about code quality.

But the techniques are still a secret![3] Peer code review has the potential to take too much time to be worth the gain in bug-fixing, code maintainability, or in mentoring new developers. The techniques that provide the benefits of peer code review while mitigating the pitfalls and managing developers' time are competitive advantages that no one wants to reveal.

Unfortunately for these successful software development organizations, we make a living making code review accessible and efficient for everyone. And that's what this book is about.

I'm interested. What next?

So code review works, but what if developers waste too much time doing it? What if the social ramifications of personal critiquing ruin morale? How can review be implemented in a measurable way so you can identify process problems?

We cover case studies of review in the real world and show which conclusions you can draw from them (and which you can't). We give our own case study of 2500 reviews. We give pros and cons for the five most common types of review. We explain how to take advantage of the positive social and personal aspects of review as well as ways managers can mitigate negative emotions that can arise. We explain how to implement a review within a CMMI/PSP/TSP context. We give specific advice on how to construct a peer review process that meets specific goals. Finally, we describe a tool that our customers have used to make certain kinds of reviews as painless and efficient as possible.

Code review can be practical, efficient, and even fun.

1 Income primarily from the "Adobe Intelligent Documents" division, defined with financial figures in Adobe Systems Incorporated Letter to Stockholders FY 2005.

2 "At the heart of our enterprise strategy are the free and ubiquitous Adobe Reader software and Adobe Portable Document Format (PDF). Adobe Reader enables users to view, print, and interact with documents across a wide variety of platforms." Ibid, page 6.

3 Some companies have published case studies on effectiveness of heavyweight inspection processes. In our experience, the overwhelming majority of code review processes are not heavyweight, and those studies are often statistically-insignificant. Details on this and our own case study are given in the essays, "Brand New Information" and "Code Review at Cisco Systems." ■

Five Types of Review

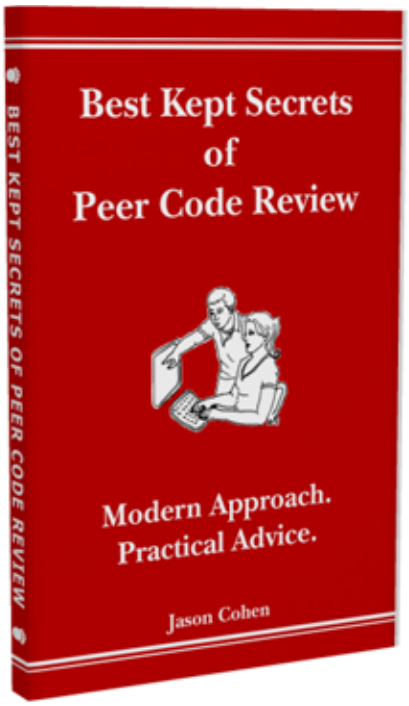
Pros and cons of formal, over-the-shoulder, e-mail pass-around, pair-programming, and tool-assisted reviews.

Author: **SmartBear Software**

basic

Reference:

This article is a single chapter of the book "Best Kept Secrets of Peer Code Review" by SmartBear Software. A compilation of 10 practical essays from industry experts, this book includes details of the largest case study on peer code review and gives specific techniques for effective reviews that your team can use right away.



SmartBear SOFTWARE

The full book can be accessed from the following link: http://smartbear.com/codecollab-code-review-book.php

There are many ways to skin a cat. I can think of four right off the bat. There are also many ways to perform a peer review, each with pros and cons.

Formal inspections

For historical reasons, "formal" reviews are usually called "inspections." This is a hold-over from Michael Fagan's seminal 1976 study at IBM regarding the efficacy of peer reviews. He tried many combinations of variables and came up with a procedure for reviewing up to 250 lines of prose or source code. After 800 iterations he came up with a formalized inspection strategy and to this day you can pay him to tell you about it (company name: Fagan Associates). His methods were further studied and expanded upon by others, most notably Tom Gilb and Karl Wiegers.

In general, a "formal" review refers to a heavy-process review with three to six participants meeting together in one room with print-outs and/or a projector. Someone is the "moderator" or "controller" and acts as the organizer, keeps everyone on task, controls the pace of the review, and acts as arbiter of disputes. Everyone reads through the materials beforehand to properly prepare for the meeting.

Each participant will be assigned a specific "role." A "reviewer" might be tasked with critical analysis while an "observer" might be called in for domain-specific advice or to learn how to do reviews properly. In a Fagan Inspection, a "reader" looks at source code only for comprehension – not for critique – and presents this to the group. This separates what the author intended from what is actually presented; often the author himself is able to pick out defects given this third-party description.

When defects are discovered in a formal review, they are usually recorded in great detail. Besides the general location of the error in the code, they include details such as severity (e.g. major, minor), type (e.g. algorithm, documentation, data-usage, error-handling), and phase-injection (e.g. developer error, design oversight, requirements mistake). Typically this information is kept in a database so defect metrics can be analyzed from many angles and possibly compared to similar metrics from QA.

Formal inspections also typically record other metrics such as individual time spent during pre-meeting reading and during the meeting itself, lines-of-code inspection rates, and problems encountered with the process itself. These numbers and comments are examined periodically in process-improvement meetings; Fagan Inspections go one step further and requires a process-rating questionnaire after each meeting to help with the improvement step.

Formal inspections' greatest asset is also its biggest drawback: When you have many people spending lots of time reading code and discussing its consequences, you are going to identify a lot of defects. And there are plenty of studies that show formal inspections can identify a large number of problems in source code.

But most organizations cannot afford to tie up that many people for that long. You also have to schedule the meetings – a daunting task in itself and one that ends up consuming extra developer time[1]. Finally, most formal methods

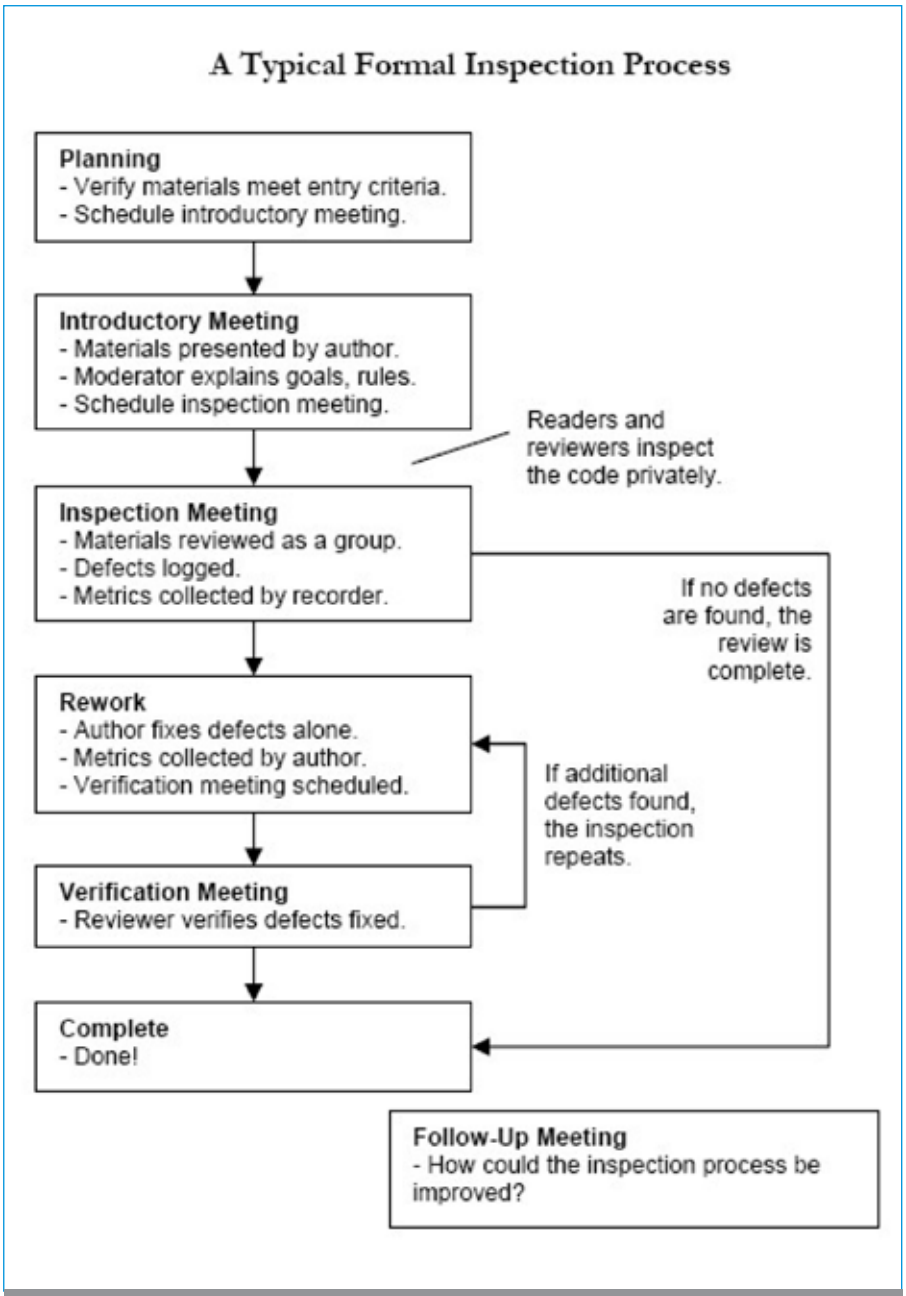


Figure 1 Typical workflow for a "formal" inspection. Not shown are the artifacts created by the review: The defect log, meeting notes, and metrics log. Some inspections also have a closing questionnaire used in the follow-up meeting.

require training to be effective, and this is an additional time and expense that is difficult to accept, especially when you aren't already used to doing code reviews.

Many studies in the past 15 years have come out demonstrating that other forms of review uncover just as many defects as do formal reviews but with much less time and training[2]. This result – anticipated by those who have tried many types of review – has put formal inspections out of favor in the industry.

After all, if you can get all the proven benefits of formal inspections but occupy 1/3 the developer time, that's clearly better.

So let's investigate some of these other techniques.

Over-the-shoulder reviews

This is the most common and informal of code reviews. An "over-the-shoulder" review is just that – a developer standing over the author's workstation while the author walks the reviewer through a set of code changes.

Typically the author "drives" the review by sitting at the keyboard and mouse, opening various files, pointing out the changes and explaining why it was done this way. The author can present the changes using various tools and even run back and forth between changes and other files in the project. If the reviewer sees something amiss, they can engage in a little "spot pair-programming" as the author writes the fix while the reviewer hovers. Bigger changes where the reviewer doesn't need to be involved are taken off-line.

With modern desktop-sharing software a so-called "over-the-shoulder" review can be made to work over long distances. This complicates the process because you need to schedule these sharing meetings and communicate over the phone. Standing over a shoulder allows people to point, write examples, or even go to a whiteboard for discussion; this is more difficult over the Internet.

The most obvious advantage of over-the-shoulder reviews is simplicity in execution. Anyone can do it, any time, without training. It can also be deployed whenever you need it most – an especially complicated change or an alteration to a "stable" code branch.

As with all in-person reviews, over-the-shoulders lend themselves to learning and sharing between developers and get people to interact in person instead of hiding behind impersonal email and instant-messages. You naturally talk more when you can blurt out an idea rather than adding some formal "defect" in a database somewhere.

Unfortunately, the informality and simplicity of the process also leads to a mountain of shortcomings. First, this is not an enforceable process – there's nothing that lets a manager know whether all code changes are being reviewed. In fact, there are no metrics, reports, or tools that measure anything at all about the process.

Second, it's easy for the author to unintentionally miss a change. Countless times we've observed a review that completes, the author checks in his changes, and when he sees the list of files just checked in he says "Oh, did I change that one?" Too late!

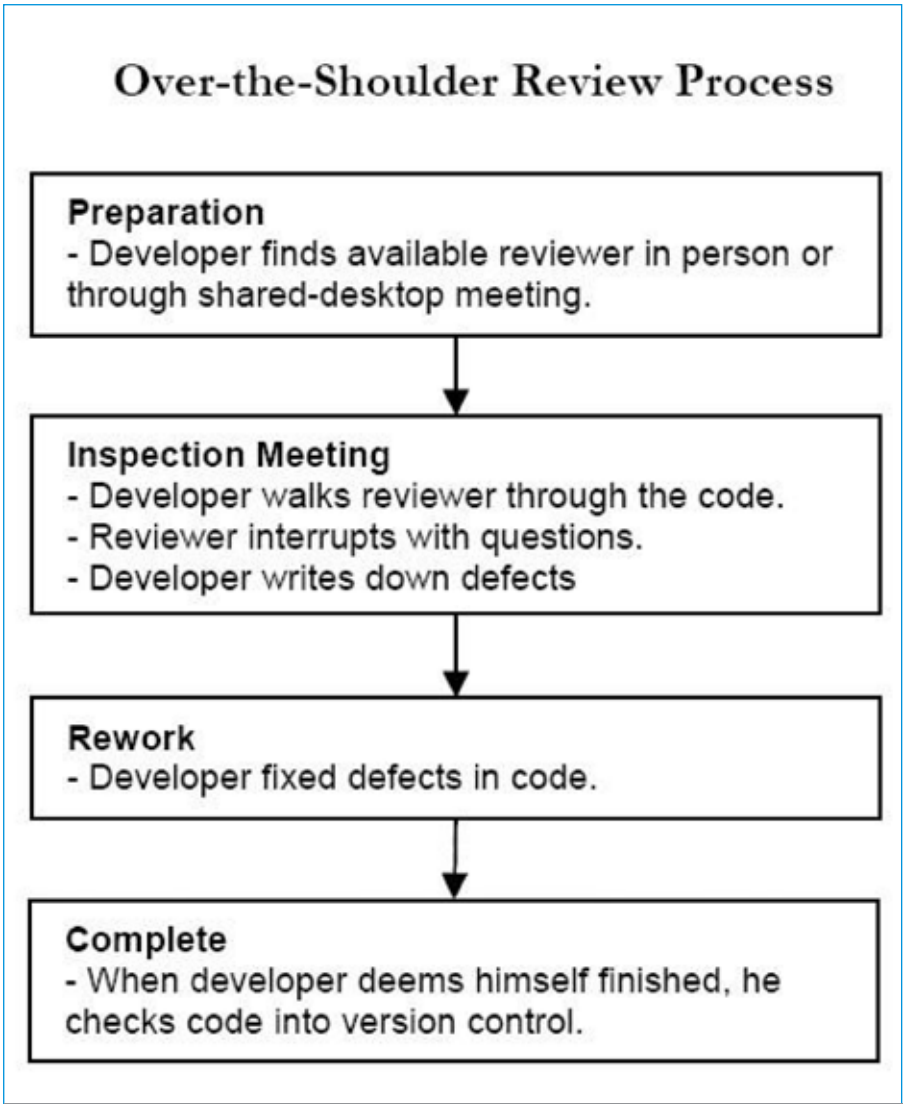


Figure 2 A typical Over-the-shoulder code walk-through process. Typically no review artifacts are created.

Third, when a reviewer reports defects and leaves the room, rarely does the reviewer return to verify that the defects were fixed properly and that no new defects were introduced. If you're not verifying that defects are fixed, the value of finding them is diminished.

There is another effect of over-the-shoulder reviews which some people consider to be an advantage, but others a drawback. Because the author is controlling the pace of the review, often the reviewer is led too hastily through the code. The reviewer might not ponder over a complex portion of code. The reviewer doesn't get a chance to poke around in other source files to confirm that a change won't break something else. The author might explain something that clarifies the code to the reviewer, but the next developer who reads that code won't have the advantage of that explanation unless it is encoded as a comment in the code. It's difficult for a reviewer to be objective and aware of these issues while being driven through the code with an expectant developer peering up at him.

For example, say the author was tasked with fixing a bug where a portion of a dialog was being drawn incorrectly. After wrestling with the Windows GUI documentation, he finally discovers an undocumented "feature" in the draw-text API call that was causing the problems. He works around the bug with some new code and fixes the problem. When the reviewer gets to this work-around, it looks funny at first.

"Why did you do this?" asks the reviewer. "The Windows GUI API will do this for you."

"Yeah, I thought so too," responds the author, "but it turns out it doesn't actually handle this case correctly. So I had to call it a different way in this case."

It's all too easy for the reviewer to accept the changes. But the next developer that reads this code will have the same question and might even remove the work-around in an attempt to make the code cleaner. "After all," says the next developer, "the Windows API does this for us, so no need for this extra code!"

On the other hand, not all prompting is bad. With changes that touch many files it's often useful to review the files in a particular order. And sometimes a change will make sense to a future reader, but the reviewer might need an explanation for why things were changed from the way they were.

Finally, over-the-shoulder reviews by definition don't work when the author and reviewer aren't in the same building; they probably should also be in nearby offices. For any kind of remote review, you need to invoke some electronic communication. Even with desktop-sharing and speakerphones, many of the benefits of face-to-face interactions are lost.

E-mail pass-around reviews

E-mail pass-around reviews are the second-most common form of informal code review and the technique preferred by most open-source projects. Here, whole files or changes are packaged up by the author and sent to reviewers via e-mail. Reviewers examine the files, ask questions and discuss with the author and other developers, and suggest changes.

The hardest part of the e-mail pass-around is in finding and collecting the files under review. On the author's end, he has to figure out how to gather the files together. For example, if this is a review of changes being proposed to check into version control, the user has to identify all the files added, deleted, and modified, copy them somewhere, then download the previous versions of those files (so reviewers can see what was changed), and organize the files so the reviewers know which files should be compared with which others. On the reviewing end, reviewers have to extract those files from the e-mail and generate differences between each.

The version control system can be of some assistance. Typically that system can report on which files have been altered and can be made to extract previous versions. Although some people write their own scripts to collect all these files, most use commercial tools that do the same thing and can handle the myriad of corner-cases arising from files in various states and client/server configurations.

The version control system can also assist by sending the e-mails out automatically. For example, a version control server-side "check-in" trigger can send e-mails depending on who checked in the code (e.g. the lead developer of each group reviews code from members of that group) and which files were changed (e.g. some files are "owned" by a user who is best-qualified to review the changes). The automation is helpful, but for many code review processes you want to require reviews before check-in, not after.

Like over-the-shoulder reviews, e-mail pass-arounds are easy to implement, although more time-consuming because of the file-gathering. But unlike over-the-shoulder reviews, they work equally well with developers working across the hall or across an ocean. And you eliminate the problem of the authors coaching the reviewers through the changes.

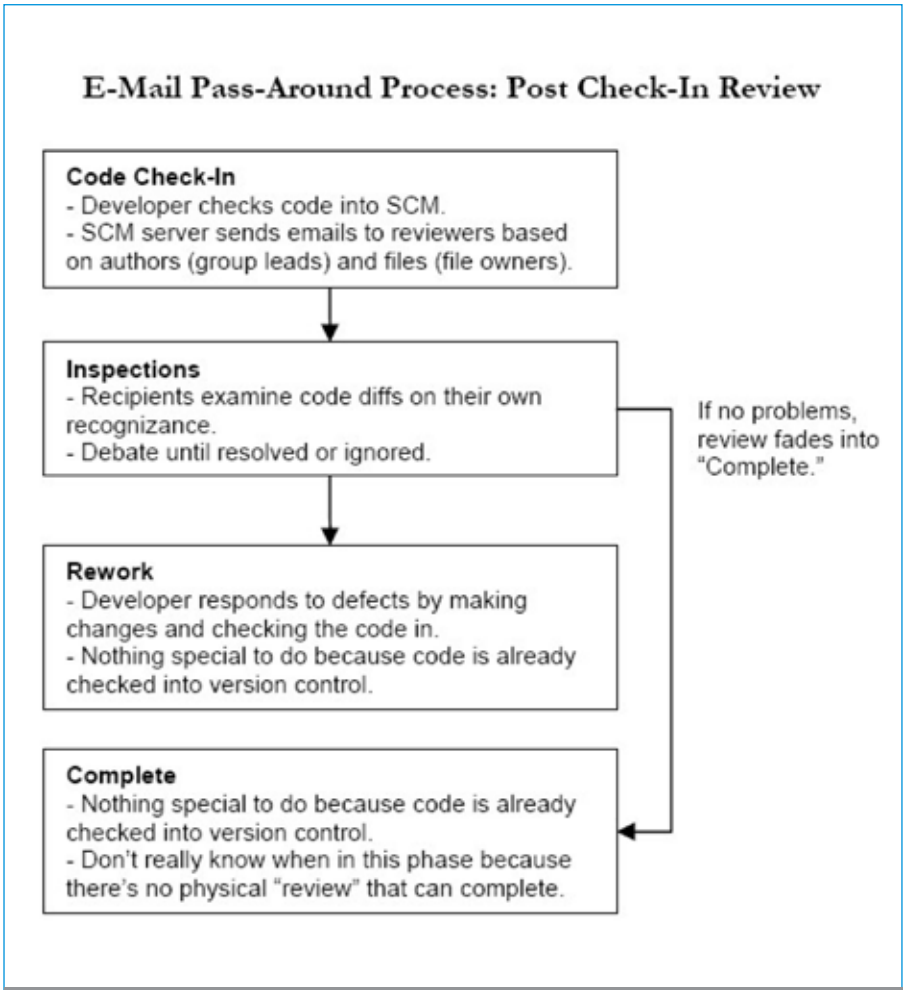


Figure 3 Typical process for an e-mail pass-around review for code already checked into a version control system. These phases are not this distinct in reality because there's no tangible "review" object.

Another unique advantage of e-mail pass-arounds is the ease in which other people can be brought into the review. Perhaps there is a domain expert for a section of code that a reviewer wants to get an opinion from. Perhaps the reviewer wants to defer to another reviewer. Or perhaps the e-mail is sent to many people at once, and those people decide for themselves who are best qualified to review which parts of the code. This inclusiveness is difficult with in-person reviews and with formal inspections where all participants need to be invited to the meeting in advance.

Yet another advantage of e-mail pass-arounds is they don't knock reviewers out of "the zone." It's well established that it takes a developer 15 minutes to get into "the zone" where he is immersed in his work and is highly productive[3]. Even just asking a developer for a review knocks him out of the zone – even if the response is "I'm too busy." With e-mails, reviewers can work during a self-prescribed break so they can stay in the zone for hours at a time.

There are several important drawbacks to the e-mail pass-around review method. The biggest is that for all but the most trivial reviews, it can rapidly become difficult to track the various threads of conversation and code changes. With several discussions concerning a few different areas of the code, possibly inviting other developers to the fray, it's hard to track what everyone's saying or whether the group is getting to a consensus.

This is even more prominent with over-seas reviews; ironic since one of the distinct advantages of e-mail pass-arounds is that they can be done with remote developers. An over-seas review might take many days as each "back and forth" can take a day, so it might take five days to complete a review instead of thirty minutes. This means many simultaneous reviews, and that means even more difficulties keeping straight the conversations and associated code changes.

Imagine a developer in Hyderabad opening Outlook to discover 25 emails from different people discussing aspects of three different code changes he's made over the last few days. It will take a while just to dig though that before any real work can begin.

For all their advantages over over-the-shoulder reviews, e-mail pass-arounds share some disadvantages. Product managers are still not sure whether all code changes are being reviewed. Even with version control server-side triggers, all you know is that changes were sent out – not that anyone actually looked at them. And if there was a consensus that certain defects needed to be fixed, you cannot verify that those fixes were made properly. Also there are still no metrics to measure the process, determine efficiency, or measure the effect of a change in the process.

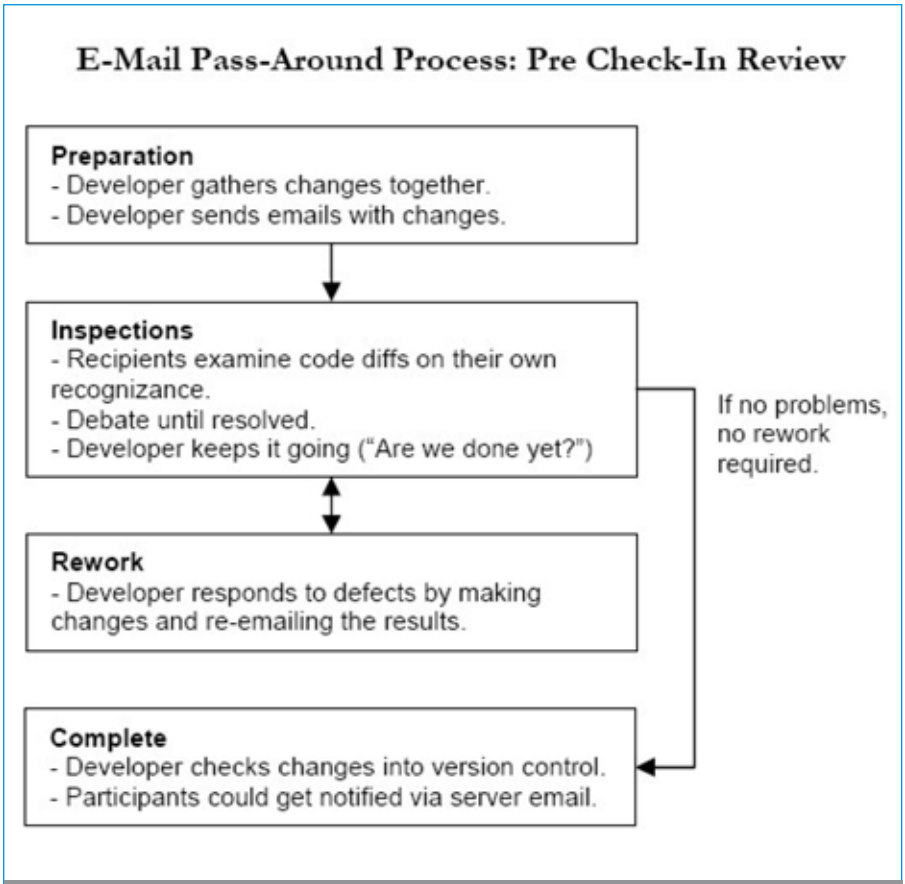


Figure 4 Typical process for an e-mail pass-around review for code already checked into a version control system. These phases are not this distinct in reality because there's no tangible "review" object.

With e-mail pass-arounds we've seen that with the introduction of a few tools (i.e. e-mail, version control client-side scripts for file-collection and server-side scripts for workflow automation) we were able to gain several benefits over over-the-shoulder reviews without introducing significant drawbacks. Perhaps by the introduction of more sophisticated, specialized tools we can continue to add benefits while removing the remaining drawbacks.

Tool-Assisted reviews

This refers to any process where specialized tools are used in all aspects of the review: collecting files, transmitting and displaying files, commentary and defects among all participants, collecting metrics, and giving product managers and administrators some control over the workflow.

There are several key elements that must be present in a review tool if it is going to solve the major problems with other types of review[4]:

Automated File Gathering

As we discussed in the e-mail pass-around section, you can't have developers spending time manually gathering files and differences for review. A tool must integrate with your version control system to extract current and previous versions so reviewers can easily see the changes under review.

Ideally the tool can do this both with local changes not yet checked into version control and with already-checked-in changes (e.g. by date, label, branch, or unique change-list number). Even if you're not doing both types of review today, you'll want the option in the future.

Combined Display: Differences, Comments, Defects

One of the biggest time-sinks with any type of review is in reviewers and developers having to associate each sub-conversation with a particular file and line number. The tool must be able to display files and before/after file differences in such a manner that conversations are threaded and no one has to spend time cross-referencing comments, defects, and source code.

Automated Metrics Collection

On one hand, accurate metrics are the only way to understand your process and the only way to measure the changes that occur when you change the process. On the other hand, no developer wants to review code while holding a stopwatch and wielding line-counting tools.

A tool that automates collection of key metrics is the only way to keep developers happy (i.e. no extra work for them) and get meaningful metrics on your process. A full discussion of review metrics and what they mean appears in the "Measurement and Improvement" essay, but your tool should at least collect these three rates: kLOC/hour (inspection rate), defects/hour (defect rate), and defects/kLOC (defect density).

Review Enforcement

Almost all other types of review suffer from the problem of product managers not knowing whether developers are reviewing all code changes or whether reviewers are verifying that defects are indeed fixed and didn't cause new defects. A tool should be able to enforce this workflow at least at a reporting level (for passive workflow enforcement) and at best, at the version control level (with server-side triggers that enforce workflow at the version control level).

Clients and Integrations

Some developers like command-line tools. Others prefer integrations with IDE's and version control GUI clients. Administrators like zero-installation web clients. It's important that a tool supports many ways to read and write data in the system.

Developer tools also have a habit of needing to be integrated with other tools. Version control clients are inside IDE's. Issue-trackers are correlated with version control changes. Similarly, your review tool needs to integrate with your other tools – everything from IDE's and version

control clients to metrics and reports. A bonus is a tool that exposes a public API so you can make customizations and detailed integrations yourself.

If your tool satisfies this list of requirements, you'll have the benefits of e-mail pass-around reviews (works with multiple, possibly-remote developers, minimizes interruptions) but without the problems of no workflow enforcement, no metrics, and wasting time with file/difference packaging, delivery, and inspection.

The drawback of any tool-assisted review is cost – either in cash for a commercial tool or as time if developed in-house. You also need to make sure the tool is flexible enough to handle your specific code review process; otherwise you might find the tool driving your process instead of vice-versa.

Although tool-assisted reviews can solve the problems that plague typical code reviews, there is still one other technique that, while not often used, has the potential to find even more defects than standard code review.

Pair - Programming

Most people associate pair-programming with XP[5] and agile development in general, but it's also a development process that incorporates continuous code review. Pair-programming is two developers writing code at a single workstation with only one developer typing at a time and continuous free-form discussion and review.

Studies of pair-programming have shown it to be very effective at both finding bugs and promoting knowledge transfer. And some developers really enjoy doing it.

There's a controversial issue about whether pair-programming reviews are better, worse, or complementary to more standard reviews. The reviewing developer is deeply involved in the code, giving great thought to the issues and consequences arising from different implementations. On the one hand this gives the reviewer lots of inspection time and a deep insight into the problem at hand, so perhaps this means the review is more effective. On the other hand, this closeness is exactly what you don't want in a reviewer; just as no author can see all typos in his own writing, a reviewer too close to the code cannot step back and critique it from a fresh and unbiased position. Some people suggest using both techniques – pair-programming for the deep review and a follow-up standard review for fresh eyes. Although this takes a lot of developer time to implement, it would seem that this technique would find the greatest number of defects. We've never seen anyone do this in practice.

The single biggest complaint about pair-programming is that it takes too much time. Rather than having a reviewer spend 15-30 minutes reviewing a change that took one developer a few days to make, in pair-programming you have two developers on the task the entire time.

Some developers just don't like pair-programming; it depends on the disposition of the developers and who is partnered with whom. Pair-programming also does not address the issue of remote developers.

A full discussion of the pros and cons of pair-programming in general is beyond our scope.

Conclusion

Each of the five types of review is useful in its own way. Formal inspections and pair-programming are proven techniques but require large amounts of developer time and don't work with remote developers. Over-the-shoulder reviews are easiest to implement but can't be implemented as a controlled process. E-mail pass-around and tool-assisted reviews strike a balance between time invested and ease of implementation.

And any kind of code review is better than nothing.

1 See the Votta 1993 case study detailed in "Brand New Information".

2 See the case study survey elsewhere in "Code Review at Cisco Systems".

3 For a fun read on this topic, see "Where do These People Get Their (Unoriginal) Ideas?" Joel On Software. Joel Spolsky, Apr 29, 2000.

4 In the interest of full-disclosure, Smart Bear Software, the company that employs the author of this essay, sells a popular peer code review tool called Code Collaborator for exactly this purpose. This product is described in the "Code Collaborator" essay in this collection; this section will discuss general ways in which tools can assist the review process.

5 Extreme Programming is perhaps the most talked-about form of agile development. Learn more at http://www.extremeprogramming.org. ■

Confrontation of developers and testers

basic

Author: ***Marina Lager and Andrey Konushin***

About the authors:

Andrey Konushin - Expert in software testing and quality assurance. ISTQB Certified Tester, Advanced Level.

Master of techniques and technologies in informatics and computer engineering, Vladimir State University, Russia. His main research interests include information management and system analysis both in software testing and related fields.

Quality assurance expert and test manager at software development companies with big experience in tens of projects. CEO of "Knowledge Department Russia" – a coaching and consulting company, offering its services worldwide.

Since 2006 senior lecturer at Information Systems & Information Management chair of Vladimir State University. Andrey developed and implemented the "Software testing foundations" course for the University based on ISTQB Foundation Level Syllabus.

One of the founders of the Russian Software Testing Qualifications Board. Since 2006 he is the President of the RSTQB.

Marina Lager - 24 years old.

Specialist in software testing. Degreed specialist in informatics and computer engineering, Vladimir State University, Russia.

Since 2007: Software test engineer at Inreco LAN, a software development company (http://inrecolan.com).
Since 2010: Acting Test Lead of Inreco LAN

Her main research interests include project management and communication between team members in software development.

In the software development process different professionals are involved: managers, analysts, developer and testers. But they are primarily people who communicate among themselves, who have formed a definite relation to each other, and who have work on a common project from day to day. But sometimes their attitude is much like confrontation. So, in order...

Developer (or programmer) - a software development specialist. Usually, the word "programmer" means person who specializes in writing the source code following the desired specifications.

Tester - a specialist, professional responsibilities of whom is to detect, localize, describe and track errors in the software. Thus, the task of the tester - a search of defects (bugs), which developers had made.

But is there a real need of testers, when developers can check their work?

The most known and popular testing certification scheme – ISTQB (International Software Testing Qualifications Board) – defines the following: "A way of thinking, needed in the process of testing or review, differs from the ways of thinking needed for programming and analysis. With the right thinking, the developers can test their own code. But division of responsibilities is usually done in order to help focus different efforts and provide additional benefits, such as an independent opinion of the trained professionals in the testing."

Definition of independent testing is given in the ISTQB glossary: "Independence of testing is separation of responsibilities, which encourages the accomplishment of objective testing."

It also defines several levels of independence:
- Tests are designed by the person who wrote the program (a low level of independence).
- Tests are designed by another person (for example, from the development team).
- Tests are designed by person from another organizational group (e.g., an independent team of testers).
- Tests are designed by person from another organization or company (e.g., outsourcing or certification by another company).

The last two levels of independence are more effective than the first. Finding of program crashes during testing may be perceived as criticism of the product or author. Therefore, testing is often seen as a destructive activity, even if it is constructive in terms of risk management. As a consequence, sometimes programmers and testers are, to put it mildly, not friendly.

For example, the programmer has implemented its mission and went to home happy. But next morning he finds many error reports in his implementation, which had taken much time and effort. Few of us want to admit their mistakes. Sometimes programmers generally think

that errors occur because of testers. Even questions like this may appear: "When testers will stop to find bugs?!". Of course, the defects do not arise because of the testers. Precisely the contrary, testing helps to improve the quality of software, but some people forget this, and begin to utter unpleasant words to tester, to write angry emails, reject defects, etc.

Now let's try to look at this situation from the other side. Most testers feel themselves just inferior people in the team. Wages of programmers usually higher. If the software works well, the developers receive the award, because it is implemented. And if a product works poorly, the blame can be at testers, because they had inspected or tested not enough well.

Add to this discontented programmers who constantly reject defects, provide functionality at the last moment, and there is no enough time for full inspection, the desire to remain true to profession of a tester is approaching to zero.

But software development process requires both developers and testers, and it is important that they have good working relations. To do this, of course, some efforts should be made.

Testers, to live peacefully with the programmers, need to learn understanding, and seek an individual approach. But besides this, tester must have a certain set of qualities to be respected by colleagues:

• Attention, because problem can arise not only due to the fact that the code has errors inside, but because tester does something incorrectly.
• Observation, as it is necessary to notice the slightest flaws in the program.
• Pedantry, as it is necessary to conduct any test carefully and not once.
• Assiduity, as it is necessary to find a mistake, try to discover its cause, describe, and then double-check.
• Tact, as indication of the error must be as gently as possible, specify only the facts and not give any personal assessment.
• Persistence, as it is often necessary to defend the rightness, to prove the existence of the defect (but "not to bend the stick" is important, too).
• Creative thinking that would come up with new tests.
• Communication skills, because of the need to collaborate within the testing team, in addition to communicate with the developers, authorities, and sometimes directly with customers.
• Desire to learn, because you need to know as much as possible about the product, the technology of its development, as well as to explore new means of testing.

Also the search for faults in the system requires curiosity, professional pessimism and experience which is based on an intuitive search for errors.

As a testing professionals we can give a few tips to all testers, especially youngest:

• Note that you and developer perceive the software in various ways: programmer usually focuses on a specific module (functions, parts), but you can imagine the entire system (as interacting components).
• Your attitude should be critical, in order to find as many mistakes as possible.
• Describing defects, try to gently, step by step explain how to reproduce the bug (if bug is not reproduced regularly, make sure you mentioned it).
• Insist on providing you with all possible documentation, as this will increase the efficiency of your work.
• Do not think of yourself as of worse or better than other team members.
• Work with great responsibility.
• Do not demand too much, i.e. if you do not have computer of the last generation, then work with what you have, but keep in mind the possible problems when planning the test.
• Focus on the software product, rather than the individual programmer, manager, analytics, etc.
• Do not forget that developers are person with their advantages and disadvantages.
• Remember that developer usually is committed to create a software that works in general. Incorrect operation of small functions, misspelling, etc. for him is not errors. In this case, try calmly and convincingly prove that it is still lack and developer needs to fix it.

But not only the tester must exert efforts and establish normal relations with developers. For developers it is desirable to keep in mind the following facts:

• The task of a tester is to find a mistake, testers think critically. But it does not mean to hurt you, it's the characteristic that is essential for creating quality software product.
• Labor of tester is complicated too, it is intellectual activity, which deserves respect. Tester wants the project to be successful, and not trying to point out your shortcomings. Tester is trying to address the problem from the point of view of the user.
• Before giving tasks and modules for testing, you should check them yourself, at least superficially - it will shorten the development time of the product as a whole. You will demonstrate yourself as a competent specialist, and possibly your relationship with testers will improve.
• For the success of the team, it is preferable in a timely manner to correct defects and to put them into the database with relevant information. Errors may block other functional verification, as well as correcting some defects may lead to others.
• If troubles, ridicule, accusations, etc., are expected for error reports, then people may not notice errors or do not make record on them (if the report is wrong, just tell testers how best to execute it, and not scold them).
• Tester sometimes can make mistakes, like all people. Do not judge them for it too seriously.

In relations between testers and developers, there is another person who controls the entire process - manager. Managers must do their best too to avoid conflicts within the team.

Hope that these recommendations will be helpful. Above all, love your work and colleagues, and remember that you are a team member working on a common goal – qualitative successful project.

1. ISTQB Standard glossary of terms used in Software Testing.
2. ISTQB Certified Tester. Foundation Level Syllabus.
3. Software Test [2009], http://s-test.narod.ru/Testing/tester.htm
4. About developers [2009], http://babruisk.com/uncategorized/pro-programmistov/12434
5. How tester must interact with developer? What are their relations? [2009], http://content.mail.ru/arch/27675/2270268.html
6. Software tester in search of defects [2009], http://work.com.ua/articles/subject/295/
7. Work of tester – grey working days [2009], http://www.it4business.ru/lib/183/
8. Who is tester [2009], http://www.software-testing.ru/library/around-testing/job/85-who-is-tester
9. Veltsman Oleg, Litovchenko Evgenia, "Constantly indicating mistakes to colleagues, testers make enemies" [2009], http://www.europa-personal.ru/profesii225.htm

Quality Management Systems, Environmental Management Systems, etc. – Are They All Informatization and Efficiency Improvement Projects or Just a Farce?

basic

Author: ***Stanislav Ogryzkov***

About the author:

30 years old. Specialist in enterprise-wide information systems, business process re-engineering (BPR), quality management (including testing). ISTQB Certifed Tester, Foundation Level, and a certified internal auditor of quality management systems (ISO 9000).
Graduate of Vladimir State University, Russia. MSc major in computer science, PhD major in technical science. One of the two first ISTQB Certifed Tester in Russia.

Since 2004: Quality Assurance Person at Inreco LAN (inrecolan.com), an offshore software development outsourcing company located in Vladimir, Russia. Since 2005: Quality Assurance Manager at the same company. Since 2006: Business Process Improvement Manager at the company. At last, since 2010, Chief Information Officer (CIO) at Inreco LAN.

See http://stanislaw.ru/eng/author/resume.asp for details.

As we all know, nowadays it is extremely popular to certify your own enterprise against numerous international standards: of quality management (ISO 9000), environmental management (ISO 14000), occupational health and safety management, etc., as well as against their branch versions (for example, CMMI is the specifying standard of quality management in software development area).

From one side, such a situation is definitely good, when all enterprises in the world work in accordance with uniform standards (world economy integration, etc.). From another side, mass certification lead to that the certification process itself became just a business, when certificates are given right and left “for a few dollars more”.

For example, a certificate of a quality management system satisfying the requirements of ISO 9000:2000 standards became a “pass” for many arrangements, tenders and markets long ago. Respectively, many enterprises nowadays need to get the certificate “as soon as possible”.

In the fullness of time I worked for an enterprise that was trying “to get the certificate quickly”, and I took active part in the preparation process and

the certification process itself. Then, finally the enterprise successfully got the certificate of its quality management system. That is, formally we met all the requirements but in fact stayed on the same low level of production process organization and its quality management.

These trends take place not only in Russia but all over the world. In point of fact, this discredits certification, nullifies its value as the guarantee of the same quality level. For example, many Indian and Chinese software development companies (mostly outsourcers) have so fishily many CMMI certificates that potential clients just do not trust them anymore!..

In the ideal case, an enterprise must start the certification process not when it needs to participate in a beneficial tender but "naturally", when it becomes **mature internally**, when its production processes achieve a proper level which will be confirmed by the certificate. In my case, I hope one day the software development company where I work now (Inreco LAN) will get a certificate of quality (e.g. ISO 9000) easily – just because internally we have already became mature enough.

So why the ideal case is not the regular case in our real life? Most of all, because of the certification price: when an enterprise wants to became internally mature for certification (not "to get the certificate a.s.a.p."), I spends **more money** on long-term work of external consultants and/or internal specialists in business process improvement, on additional hardware and software, including huge systems like ERP, their installation and implementation, on rising personnel's qualification, etc. Moreover, the real, honest certification require to spend **more time**: on surveys, business process re-engineering and optimization, on creating the appropriate enterprise standards, their enforcement and improvement, etc. – generally, on the elaboration of a high production culture.

Of course, some short-sighted top managers prefer instant benefit by winning a tender (that required to have a certificate) rather than long-and-costly certification results. However, we need to remember that all these management systems were "invented" not for fun but to bring real advantages for enterprises by improving their production efficiency.

P.S. The sign like

that is the symbol of a certified quality management system must not be printed on production items as the symbol of these items' quality because it points to the quality of the production process, not to the quality of its results (though it is expected that a qualitative process produces qualitative things). ■

Requirements
Engineering
Qualifications Board

REQB
show your competence become certified!

Many problems in the field of Software Quality occur because of bad requirements. REQB is setting international standards for Requirements Engineering to change that!

www.reqb.org