METHODS & TOOLS

Global knowledge source for software development professionals **ISSN 1661-402X**

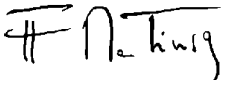
Winter 2005 (Volume 13 - number 4)

Collective Ownership of Software

As software developers, we tend to consider the programs we develop as our personal properties. This could be influenced by how we learned to develop software. As a student or at home, we created for ourselves a piece of code to say "Hello World". Sometimes you can continue to develop software exclusively for yourself, but in most of the cases you are soon confronted to the reality that your software will concern other people: developers, testers, users, maintainers. Even when you stay on your own, you will have a different look at your programs six months after having written them.

The fact that "our" code will have its own life is an important fact that we should take into consideration when we develop it. When we choose the name of a variable, when we put comments, when we make an architectural choice, we don't have to consider only our own short-term needs, even if it is quicker to type "a" than "account". We have to think that other persons will be influenced by these decisions. This should not prevent us to be proud of our work, but we have to respect the other actors when we create it. It is not always easy to have a good balance between personal and collective ownership. If a module doesn't seem to work, the first reaction is often to go to the author of the code and to say: "there is a problem with **your** code". And the programmer's first reaction will often to think that **somebody else** could have modified the code or wrongly integrated it, because we would often like to minimise the probability of our own failures. We have to acknowledge that the value of our code depends on our capacity to integrate it in a system, often with a long-term perspective. I think that success can be improved by the creation of a culture that rewards the individual contributions and, at the same time, makes clear that everybody should work together to create a common solution.

In our mostly capitalist organisations, it could seem ironic that the success of our personal development activities could be linked to the collective ownership, not of production means like proposed by Karl Marx, but of software applications. A large part of the value of our code depends on what others can do with it.



Inside

Choosing and Managing the Ideal Test Team

Lloyd Roden Lloyd@grove.co.uk
Grove Consultants, www.grove.co.uk

Introduction

"People are the most important asset of any company"

Do we agree with this statement? The majority of us would answer "yes", the main exception would be if we owned a fully automated robotics company! It is the people that make the company what it is. People are important; we therefore need to invest time in people as well as tasks related issues.

Creating the right balance between task issues and people issues is vital for successful test management. Focussing on just task issues can make us little more than a military camp where people are accustomed to follow orders. Whereas if we focussed only on the people's needs and ignore tasks that must be accomplished then we move closer towards the 'holiday camp' scenario! This balance is not only important it is also very difficult to do well. In this paper we take a look at some of the *people related issues* in software testing.

While it is often said that 'anyone can test', the skills and makeup of the test team are important and must be managed and cultivated properly. We shall have a look at some of the key ingredients in building (and retaining) successful test teams within our organisation. Building a test team is one thing but keeping and maintaining a healthy, effective and efficient test team is quite a different matter.

During my 16 years career as a test manager / test consultant, I have found that managing people is often one of the most difficult things to do well. If this is the case then why do we spend more time dwelling on the task issues? We shall take a look at some of the key ingredients of a successful test team and some of the problems we can encounter preventing us becoming a successful team and how we might overcome them.

We will also use the "tester's style analysis questionnaire" to discover the 4 types of tester that exist within our organisation; the pragmatist, the facilitator, the analyst and the pioneer. It is important to recognise differences that exist so that we can maximise their strengths rather than dwell on their weaknesses. The analysis questionnaire can also be used to identify how conflicts arise and how to defuse "explosive" situations. Once a team has been formed it is important to motivate the team. I shall explain my top four tips for motivating our testers to help them become passionate and committed to a career in testing.

Know Your Team

Tom DeMarco and Tim Lister in their book "*Peopleware*" describe how we can create an atmosphere for *Jelled Teams* - teams that produce "success" and "productive harmony". So what are some of the common distracters when it comes to forming successful teams?

Physical separation. It is difficult sometimes to always have the test team in close proximity to the rest of the project team. But I have found that the more separation there is, the more problems transpire between the teams. It is important – where possible to have the test team, development team and designers located as close as possible.

Being unfair. Human nature is to crave fairness. From a very early age children seek fairness from their parents. I have two children and even now when both of them are in their 'teens', I still need to be fair. This human nature is in adulthood as well. The general rule for unfairness is when favouritism is shown to one group above the other. This can manifest itself in a number of different ways; salary, remuneration, office space, paid overtime and other financial and non-financial rewards.

Communication breakdown. Good communication is one of the key ingredients in any relationship. This is vital within the test team. To communicate with key project personnel with regards the test assessment enabling them to make informed decisions.

No common goals. The test team must have a vision – clear direction from the management in order to be productive. Otherwise the team will end up heading in different directions. State the objectives of the test team and seek approval of these with senior management. Some examples of good test objectives are:

- *assess software quality*
- *help achieve software quality*
- *assess and report on risk*
- *help preserve software quality*

Duplication of effort. One of the most frustrating activities in testing is the feeling of "*déjà vu*" – that someone has already performed the tests that we are running. Duplicating testing can be annoying to all parties – the feeling that we are wasting our time, which is a luxury we often do not have in testing! Simple techniques like reviews, allocation of tests and paired testing can alleviate this problem. Time spent planning who does what at the start of the project is time well spent.

Lack of management support. If the test team feel as though management do not support the test activity then this can have a severely damaging affect on the team. Some of the key indicators where support is not being shown:

- *not being listened to*
- *test results not being acknowledged*
- *test estimates being ignored*
- *very little of no tool support*

Adopting a blame culture. From a very early age we like to blame. If you don't believe me then take some time watching the activities of small children. When one child does something wrong then the tendency is to blame the others – usually to avoid punishment! Why do we blame? Because we usually want others to feel bad! Why is a blame culture unhealthy for productive test teams? Because we become fearful of taking any risk in case we make a mistake.

We went into one client and asked "do you operate a blame culture?" and they replied; "well if we do…it's their fault". Another client relied; "no we operate a revenge culture!" If we are to learn, progress and become more productive as a team then we must fight the "blame culture" mentality.

Failure to appreciate. Everyone wants to be acknowledged and appreciated for the work they do within the organisation. If we do not begin to appreciate one another then people feel as though they are 'worthless'.

Take time to watch your team and start to praise them for good work. It is important that this activity is not forced, but is born out of a natural relationship in the team. Rewards are sometimes good, however a simple “thank you” or “well done” goes a long way!

The key components of a ‘jelled’ test team:

- Trust and support
- Good communication
- Strong leadership
- Identity – having a sense of direction
- Building a sense of belonging to an elite
- Providing a lot of satisfying closures
- A move from independence to inter-dependence
- Recognition of strengths within the team

The tester’s style analysis

Based upon the communication styles questionnaire I have noticed that there are 4 types of tester that exist within our organisation. The questionnaire has been adapted to help assess these types and the type of testing work that these styles enjoy. This enables us to manage our teams more effectively. Assigning correct work to the correct type is essential for greater productivity.

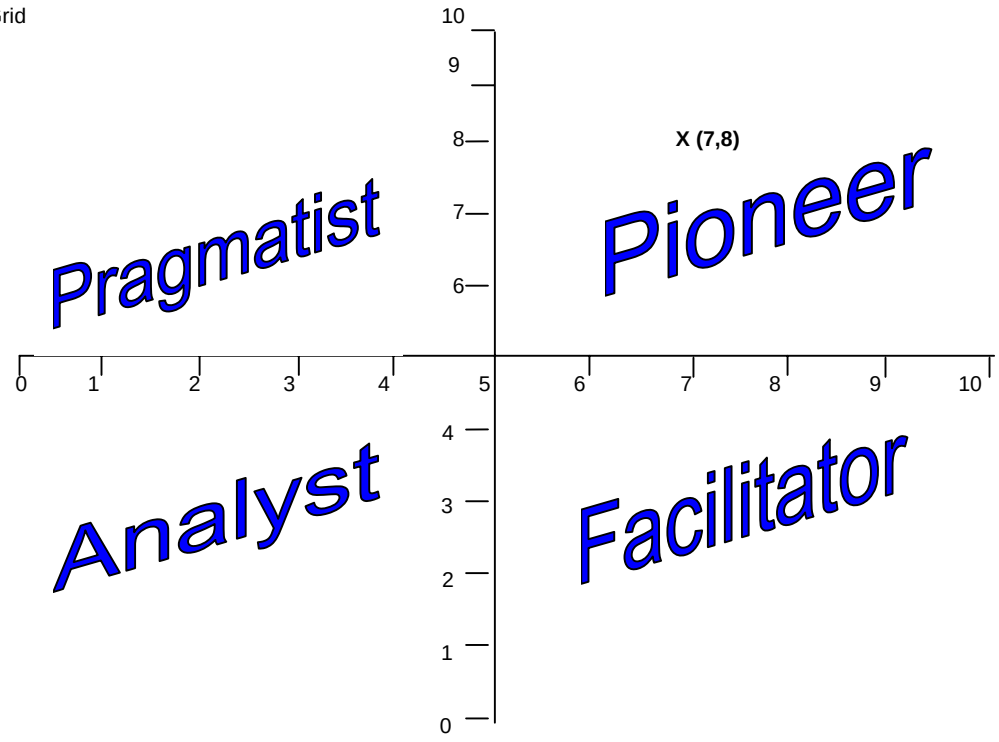
The questionnaire:

Tester's Style Analysis

NAME	A Tester

X-Axis				Y-Axis			
Friendly	x	Formal	__	To the point	x	Indirect	__
Approachable	x	Retiring	__	Challenging	x	Accepting	__
Casual	__	Business Like	x	Quick	x	Leisurely	__
Open	x	Guarded	__	Insistent	__	Thoughtful	x
Unstructured	__	Organised	x	Lively	x	Relaxed	__
Sociable	x	Introvert	__	Impatient	x	Patient	__
Intuitive	__	Logical	x	Adventurous	__	Cautious	x
Random	x	Focused	__	Confronting	x	Receptive	__
Warm	x	Cool	__	Competitive	x	Co-operative	__
Perceptive	x	Insensitive	__	Strong Minded	x	Analytical	__
	7				**8**		

Grid



Based upon "The Communication Styles Analysis" by Carol Roome

How to complete the questionnaire:

1. Complete the questionnaire by marking one column in each of the x and y axis set of questions. For example I think I am more "friendly" than "formal". As illustrated above.
2. Add up the left most column for each axis. In the example above we have scored X= 7 and Y = 8. This gives a grid coordinate (7,8) which can be plotted on the grid.
3. Each grid coordinate represents a "style"
   Top Left: ***PRAGMATIST***
   Top Right: ***PIONEER***
   Bottom Left: ***ANALYST***
   Bottom Right: ***FACILITATOR***

The Pragmatist

Likes	**Dislikes**
strategic / goals **positive** **results / brief** **practical** **efficiency** **tasks**	**indecision** **vagueness** **time-wasting** **unproductive**
The '*Pragmatic*' style tester will… 1. be good for setting and monitoring short/long term goals for the team 2. be good at documenting factual 'test reports' 3. remain positive through pressure 4. be keen to adopt 'Most Important Tests' first principle 5. be a strong driving force - ensure a task is done 6. want to implement efficiency into the team 7. be self-motivated and task oriented 8. will make quick decisions **9.** enjoy challenging testing tasks	

The Pioneer

Likes	**Dislikes**
new / ideas **change** **openness** **results/efficiency** **involving others** **risks**	**standards** **detail** **'norm'** **paper-work**
The '*Pioneer*' style tester will… 1. be good at 'ad-hoc' testing / bug hunting / error-guessing/ exploratory testing 2. be good at challenging and improving things to make more efficient and effective 3. enjoy "GUI" type testing/lateral tester 4. have good ideas 5. be good at brainstorming Test Conditions 6. share ideas about different ways to approach testing 7. identify and take necessary risks when required **8.** have creative test ideas - how to find more faults	

The Analyst

Likes	Dislikes
accuracy **attention to detail** **proof** **standards** **reliable** **all alternatives**	**new / change** **untested / risks** **brief / speed** **letting go**
The '*Analysing'* style tester will… 1. be good at defining and documenting test cases 2. be good at producing test standards and procedures 3. analyse problems and finding root cause 4. produce work which is accurate and complete 5. enjoy logical tests scenarios 6. provide proof when faults are found 7. document thorough test reports 8. complete work regardless of what it takes **9.** challenge requirements	

The facilitator

Likes	Dislikes
networking **positive** **team oriented** **consensus / sharing** **building bridges** **status quo**	**pressure / deadlines** **confrontation** **isolation** **dictated**
The '*Facilitating'* style tester will… 1. be good in a RAD environment or a 'buddy' test team 2. often ask opinion before raising issues 3. be good at documentation 4. co-operate well with other departments 5. often see the 'other side' 6. be good at defusing 'us' v 'them' syndrome 7. be popular 8. make things happen - eventually! **9.** will provide support in testing to other team members	

Tester style patterns

We usually operate within a boundary and can fluctuate within that boundary. The more prominent we are within a style the more difficult it is to adapt to one of the other styles.

If we find ourselves on the line or in the centre, then we are very flexible within the styles, but can be difficult to manage.

Opposites repel!

The key in using this analysis questionnaire is to understand where we are and also where the rest of the team are so that we can assign work which will compliment their style. However opposites repel and this maybe a reason that tension exists between team members.

For example: John is a 'Pioneering' style tester and wants new ways of doing things, to help with efficiency. John is constantly challenging standards and procedures. Chris however is an 'Analysing' style tester and requires structure in order to work efficiently. Chris is often seen as challenging John for his cavalier approach to testing! Tension often exists between John and Chris – is this wrong? No, it is just that they are different.

As managers we must recognise the team's strengths as individuals as well as the team as a whole. Yes, we must address the weaknesses but we will be far more motivational if we concentrate on the strengths.

As a general principle
- *analysts & pragmatists tend towards 'task issues '*
- *facilitators & pioneers tend towards 'people issues'*

Recruiting the right people

Recruiting the right person to join the test team can be quite daunting as well as difficult. What should we look for on a CV? How do we recognise good testers during interviews? What should we do when we have no choice in the recruitment process? Whilst anyone can run tests, not everyone is a good tester and we should challenge statements that suggest "anyone can test"!

Adding one person can disrupt the group dynamics of our team. Will the candidate gel with the rest of the team?

Have you ever thought about producing a tester's aptitude test or a small application to test to see how good the potential candidate is? If you don't want to produce your own then send an email to myself and I shall send you "Grove Consultants" versions of the above.

Motivating your team

Understanding motivation

Motivated testers will be more productive. What are the key signs of our testers being motivated and more importantly – how do we recognise when they are de-motivated?

Before we look at the key motivational areas for testers we should ask ourselves four basic questions:

1. What is motivation?

The Dictionary's definition of motivation is *"to cause someone to act in a certain way"* Motivation is the will to act. It was once assumed that motivation had to be injected from the outside but it is now understood that everyone is motivated by several different forces.

2. Why is motivation important?

For the employee, the chief advantage is job satisfaction. For the employer, it can mean good quality work. Motivation encourages higher productivity in the organisation.

Signs of motivation	Signs of de-motivation
high performance	apathy & indifference
drive & enthusiasm	dissatisfaction
co-operation in overcoming problems	poor time-keeping/high absenteeism
	resists change
keen to achieve results	exaggeration of disputes
accept responsibility	generally uncooperative
working long hours!	blame
happiness & enjoyment	withdrawal
welcomes change	

3. Whose responsibility is it to motivate?

Motivation should not be left to the manger. It is everyone's responsibility to motivate! Self-motivation is however longer-lasting, we should therefore encourage self-motivated staff further by trusting them to work on their own initiatives and encourage them to take responsibility for entire tasks.

4. Should motivation be long or short term?

Motivation should be both short term and long term.

There have been numerous studies on motivation and various theories produced. To help with the understanding of motivation further we shall take a brief look at two such theories: Hertzberg's theory and Maslow's theory.

Herzberg's Theory

Frederick Herzberg, contributed to human relations and motivation in terms of his theory of motivation. Herzberg suggested that job satisfaction is mainly caused by 'motivators' and job dissatisfaction is mainly caused by 'hygiene factors'. The first part of the motivation theory involves the hygiene theory and includes the job environment. The hygiene factors are mainly external and include

- the company,
- its policies and its administration,
- the kind of supervision which people receive while on the job,
- working conditions
- interpersonal relations,
- salary, status and security.

These factors do not lead to motivation but without them there is dissatisfaction. The second part of the motivation theory involves what people actually do on the job. The motivators are

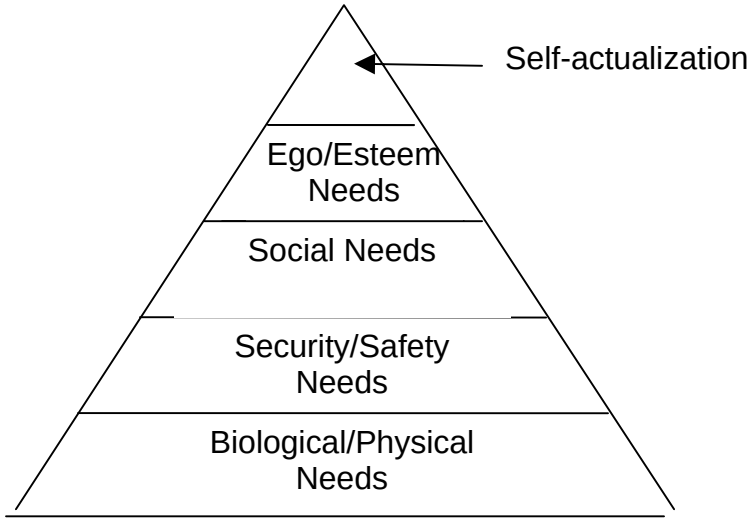
- achievement,
- challenge
- recognition,
- responsibility
- growth / advancement and
- interest in the job.

These factors result from internal generators in employees, yielding motivation rather than movement.

Both these approaches (hygiene and motivation) must be done simultaneously. Treat people as best you can so they have a minimum of dissatisfaction. Use people so they get achievement, recognition for achievement, interest, and responsibility and they can grow and advance in their work.

Maslow's theory

In the late 1960's Abraham Maslow developed a hierarchical theory of human needs. Maslow focused on human potential, believing that humans strive to reach the highest levels of their capabilities.



1. **Biological / Physiological Needs.** These needs are biological and consist of the need for oxygen, food, water, and a relatively constant body temperature. These needs are the strongest because if deprived, the person would die.

2. **Security / Safety Needs.** Except in times of emergency or periods of disorganisation in the social structure (such as widespread rioting) adults do not experience their security needs. Children, however often display signs of insecurity and their need to be safe.

3. **Social (Love, Affection and Belongingness) Needs.** People have needs to escape feelings of loneliness and alienation and give (and receive) love, affection and the sense of belonging.

4. **Ego / Esteem Needs.** People need a stable, firmly based, high level of self-respect, and respect from others in order to feel satisfied, self confident and valuable. If these needs are not met, the person feels inferior, weak, helpless and worthless.

5. **Self-actualisation/Fulfilment.** Maslow describes self-actualisation as an ongoing process. Self-actualising people are involved in a cause outside their own skin. They are devoted, work at something, something very precious to them.

Maslow set up a hierarchical theory of needs in which all the basic needs are at the bottom, and the needs concerned with man's highest potential are at the top. The hierarchic theory is often represented as a pyramid (or a set of steps), with the larger, lower levels (steps) representing the lower needs, and the upper point representing the need for self-actualisation. Each level of the pyramid (step) is dependent on the previous level. For example, a person does not feel the second need until the demands of the first have been satisfied.

Key motivators for testers

There are a number of aspects in addition to the standard motivators that are specific for testers:

Clear goals and vision for testing. It is important to know where we are heading in testing and that this is agreed with senior management. Discuss the ‘terms or reference for testing’ with your team, produce a one page document and publicise it. This shows commitment towards testing.

Support for your testers. It is important, as a test manager, that we listen to the team and if the need arises – we fight the 'tester's corner'. Discuss various courses and staff development with your team. Think about sending them on testing courses, conferences or testing seminars to improve their skill.

It is also important that we are seen to strike the right balance between 'hands-off' and 'hands-on' test management. Small things like sitting with the testers rather than in a large office and actually helping with testing will motivate the team. It shows them that you are involved rather than removed.

Promoting the value of testing. Another way we can motivate the team is to promote the value of testing at every opportunity. This is such an important aspect because so often testers are not valued for the work they do. The primary reason, I believe, is that we do not produce anything that is 'tangible'. We must constantly reflect and report on how individuals, as well as the whole team, have added value to the project and company.

Career path & Salary acknowledged. Salaries for testers should reflect their skill and the value that they add to the company. There should be no differentiation between tester, developer and designer's salary structures.

One of Maslow's motivational needs is 'self realisation' – an opportunity to improve in the job we find ourselves in. Therefore a career path for testers is essential if we are going to meet this need. Unfortunately many organisations do not have a clearly defined career path for the testers. There are a number of alternative career models we can look at:

- **Hierarchical**, where the structure starts with a trainee tester and progresses to team leader and test manager. The problem with this model is that 'management' should not be the only option in reaching the top.
- **Functional**, where the career path revolves around functions. A technical path, an automation path, a test design and analysis path and a team leader/test manager path.
- **The career cube by Gitek/Sogeti…**

The test career cube by Gitek/Sogeti *(used with permission)*

To professionalise testing within an organisation testers need to have the possibility for balanced growth in their career. This paragraph will show a method to appoint career paths for testing. Training and education, development of working experience and a coaching programme are essential components. These are addressed per function and per level in the 'career cube' (see figure 1). The career cube is a tool that is developed to provide better guidance by a personnel manager in career growth. It helps to adjust the demand from the organisation, the available knowledge and skills and the ambition of the test professionals to each other.

The three dimensions of the 'career cube' are defined as follows:

- height: functional growth
- width: functional differentiation
- depth: knowledge and skills

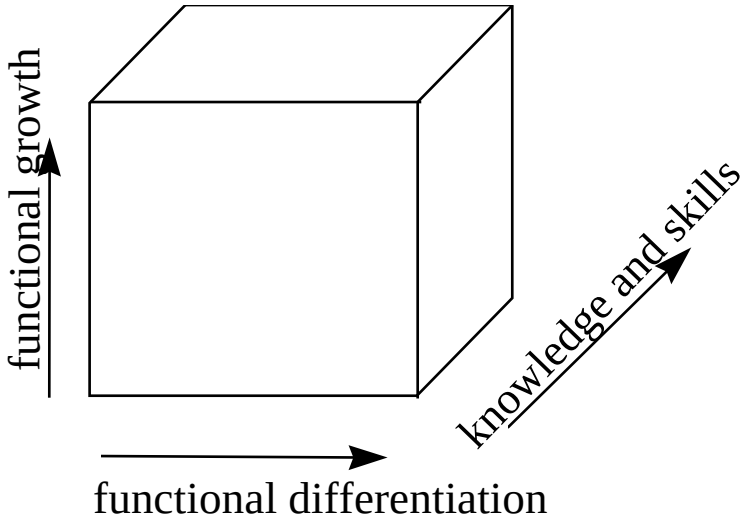


Figure 1: Dimensions of career cube

Functional growth

Functional growth implies the career an employee can make from tester to test manager. A distinction is made between vertical growth and horizontal growth. The moment vertical growth (higher function level) is no longer an option, it is possible to grow horizontally ('deeper' within the same function level).

Vertical growth leads to a higher function level, horizontal growth leads to a higher performance level within the same function level. An improvement in the conditions of employment can be realised in this structure by achieving a higher function level or a higher performance level.

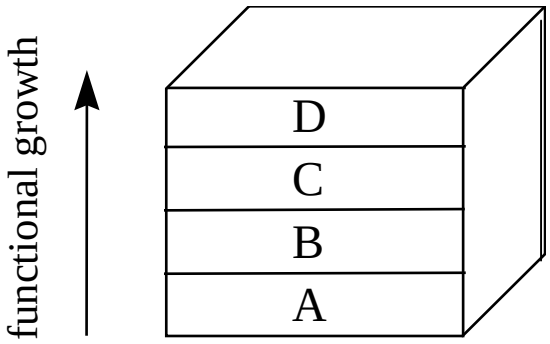


Figure 2: Functional growth

Functional differentiation

Three main streams are recognised for the dimension of functional differentiation:

- Team and project leading
  People who have an interest and the talent for managing a test project can choose for this stream.
- Methodical support
  This stream is for those who want to provide test advice and support, e.g. for the setting up of a test strategy, the selection of test specification techniques.
- Technical support
  People who have the affinity with the technical side of testing can provide technical test advice and support. Examples here are selecting and implementing test tools and setting up the technical infrastructure for testing.

Employees can choose for either stream, depending on their interest and talents.

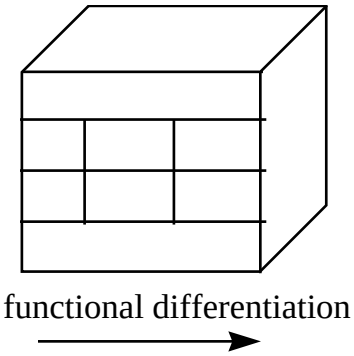


Figure 3: Functional differentiation

The diagram below shows examples of function levels with the corresponding function names for each differentiation stream.

D	general test management		
C	test project leading	methodical test advice	technical test advice
B	test team leading	methodical test specialist	technical test specialist
A	test execution		

At level A, there is no real differentiation. At these function levels broad experience is gained in the area of test execution. For levels B and C there is a differentiation. At level D there is no differentiation anymore. Employees at this level are expected to be able to successfully perform or manage in all three differentiations.

Knowledge and skills

The third dimension of knowledge and skills comprises the following components:

- (test)training;
- social skills;
- experience;
- coaching and support.

Training in testing, coaching and support are essential conditions for a sound career growth, especially for the lower function levels. Each employee is supported in a number of ways, provided by the personnel manager, the test manager and/or more experienced colleagues. A coach provides extra support to starting personnel in the field of testing, who takes care of all aspects of education and the progress therein. At higher function levels the experience and social skills are becoming increasingly important.

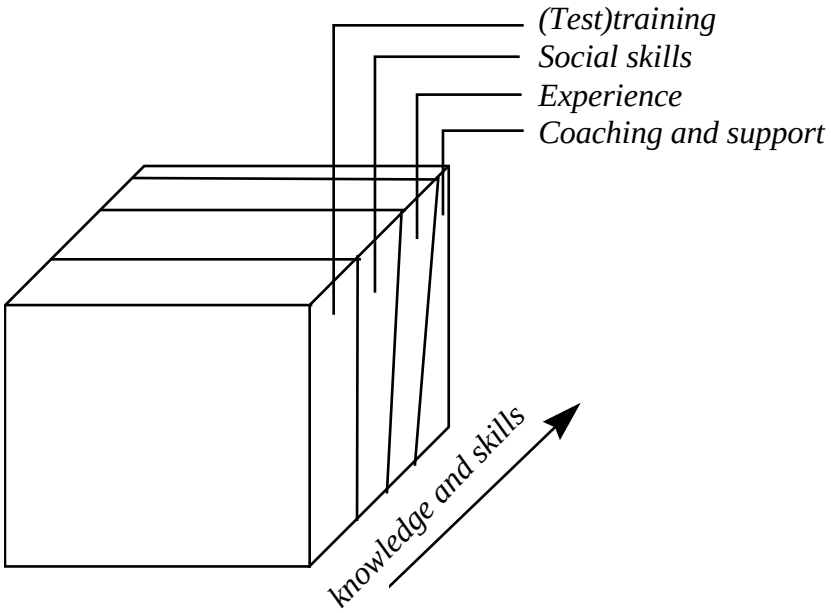


Figure 4: Knowledge and skills

Based on the three dimensions mentioned above, the 'career cube' can be filled in: for each function level the required knowledge and skills can be defined per functional differentiation.

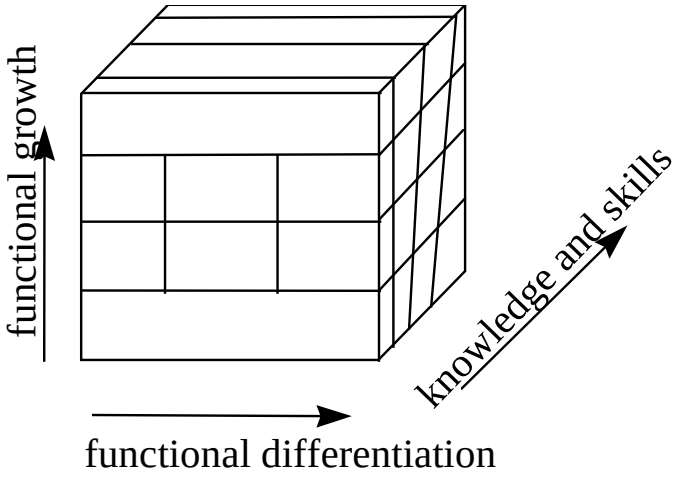


Figure 5: Complete career cube

For further details on the career cube contact Sogeti/Gitek in The Netherlands. Website www.sogeti.nl

Conclusion

In order to retain good quality testers within our test teams we must spend time developing the people side of test management. Promote the value of testing within your organisation; support your testers by seeking to introduce a healthy career structure for testers. Recognise their current strengths and try to address the skill gaps.

The test team should provide an environment for testers to grow in their skill. We should seek to motivate our staff at every opportunity and encourage 'self motivation'. Our test team should not be likened to a military camp nor should it be a holiday camp. We should try to strike a good balance between the two.

Risk Based Testing, Strategies for Prioritizing Tests against Deadlines

Hans Schaefer, Software Test Consulting, hans.schaefer@ieee.org
http://home.c2i.net/schaefer/testing.html

Often all other activities before test execution are delayed. This means testing has to be done under severe pressure. It is out of question to quit the job, nor to delay delivery or to test badly. The real answer is a prioritization strategy in order to do the best possible job with limited resources.

Which part of the systems requires most attention? There is no unique answer, and decisions about what to test have to be risk-based. There is a relationship between the resources used in testing and the risk after testing. There are possibilities for stepwise release. The general strategy is to test some important functions and features that hopefully can be released, while delaying others.

First, one has to test what is most important in the application. This can be determined by looking at visibility of functions, at frequency of use and at the possible cost of failure. Second, one has to test where the probability of failure is high, i.e. one may find most trouble. This can be determined by identifying especially defect-prone areas in the product. Project history gives some indication, and product measures like complexity give more. Using both, one finds a list of areas to test more or less.

After test execution has started and one has found some defects, these defects may be a basis for re-focussing testing. Defects clump together in defect-prone areas. Defects are a symptom of typical trouble the developers had. Thus, a defect leads to the conclusion that there are more defects nearby, and that there are more defects of the same kind. Thus, during the latter part of test execution, one should focus on areas where defects have been found, and one should generate more tests aimed at the type of defect detected before.

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

Introduction

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

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

In this paper we are talking about the higher levels of testing: integration, system and acceptance test. We assume that developers have done some basic level of testing of every

program (unit testing). We also assume the programs and their designs have been reviewed in some way. Still, most of the ideas in this paper are applicable if nothing has been done before you take over as the test manager. It is, however, easier if you know some facts from earlier quality control activities such as design and code reviews and unit testing.

1. The bad game

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

How to get out of the game?

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

One strategy is to find the right quality level. Not all products need to be free of defects. Not every function needs to work. Sometimes, you have options to do a lot about lowering product quality. This means you can cut down testing in less important areas.

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

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

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

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

2. Understanding necessary quality levels

Software is embedded in the larger, more complex business world. Quality must be considered in that context (8).

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

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

You have to be sure which qualities and functions are important. Fewer defects do not always mean more profit! You have to research how quality and financial performance interact. Examples of such approaches include the concept of Return on Quality (ROQ) used in corporations such as AT&T (9). ROQ evaluates prospective quality improvements against their ability to also improve financial performance. Be also aware of approaches like Value Based Management. Avoid to fanatically pursuing quality for its own sake.

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

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

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

Risk is the product of damage and probability for damage to occur. The way to assess risk is outlined in figure 1 below. Risk analysis assesses damage during use, usage frequency, and determines probability of failure by looking at defect introduction.

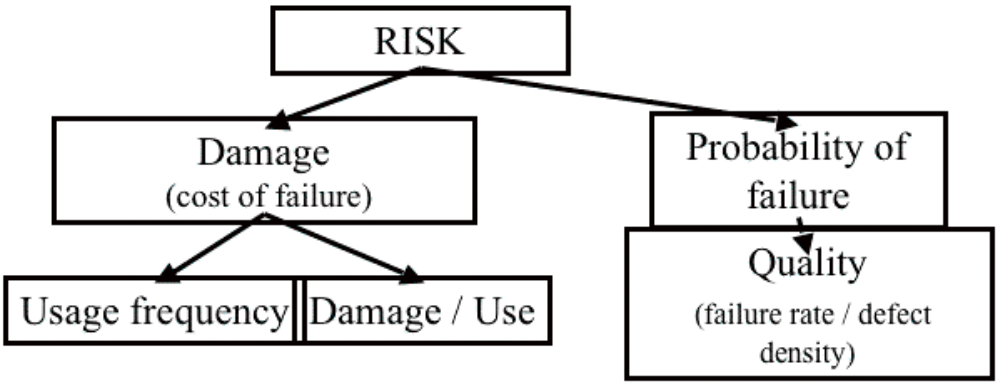


Figure 1: Risk definition and structure

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

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

When dealing with all the factors we look at, the result will always be a list of functions and properties with an associated importance. In order to make the final analysis as easy as possible, we express all the factors in a scale from 1 to 5. Five points are given for “most important” or “worst”, or generally for something having higher risk, which we want to test more, 1 points is given to less important areas. (Other publications often use weights 1 through 3).

The details of the computation are given later.

3.1. Determining damage: What is important?

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

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

Major factors include:

- Critical areas (cost and consequences of failure)

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

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

  Output that is immediately needed during working hours is more critical than output that could be sent hours or days later. On the other hand, if large volumes of data to be sent by mail are wrong, just the cost of re-mailing may be horrible. The damage may be classified into the classes mentioned down below, or quantified into money value, whatever seems better. In systems with large variation of damage it is better to use damage as absolute money value, and not classify it into groups.

  A possible hierarchy for grouping damage is the following:

  A failure would be catastrophic (3)

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

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

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

A failure would be damaging (2)

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

A failure would be hindering (1)

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

A failure would be annoying (0)

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

- Visible areas

The visible areas are areas where many users will experience a failure, if something goes wrong. Users do not only include the operators sitting at a terminal, but also final users looking at reports, invoices, or the like, or dependent on the service delivered by the product which includes the software. A factor to take into account under this heading is also the forgivingness of the users, i.e. their tolerance against any problem. It relates to the importance of different qualities, see above.

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

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

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

- Usage frequency

Damage is dependent on how often a function or feature is used.

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

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

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

A possible hierarchy is outlined here (from (3)):

Unavoidable (3)

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

Frequent (2)

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

Occasional (1)

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

Rare (0)

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

An alternative method to use for picking important requirements is described in (1).

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

3.2. Failure probability: What is (presumably) worst

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

- Complex areas

  Complexity is maybe the most important defect generator. More than 200 different complexity measures exist, and research into the relation of complexity and defect frequency has been done for more than 20 years. However, no predictive measures have until now been generally validated. Still, most complexity measures may indicate problematic areas. Examples include long modules, many variables in use, complex logic, complex control structure, a large data flow, central placement of functions, a deep inheritance tree, and even subjective complexity as understood by the designers. This means you may do several complexity analyses, based on different aspects of complexity and find different areas of the product that might have problems.

- Changed areas

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

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

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

- Impact of new technology, solutions, methods

  Programmers using new tools, methods and technology experience a learning curve. In the beginning, they may generate many more faults than later. Tools include CASE tools, which

may be new in the company, or new in the market and more or less unstable. Another issue is the programming language, which may be new to the programmers, or Graphical User Interface libraries. Any new tool or technique may give trouble. A good example is the first project with a new type of user interface. The general functionality may work well, but the user interface subsystem may be full of trouble.

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

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

- Impact of the number of people involved

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

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

Care should be taken in that analysis: Some companies (11) employ their best people in more complex areas, and less qualified people in easy areas. Then, defect density may not reflect the number of people or their qualification.

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

- Impact of turnover

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

- Impact of time pressure

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

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

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

- Areas which needed optimizing

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

- Areas with many defects before

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

- Geographical distribution

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

  - People having their offices in different floors of the same building will not communicate as much as people on the same floor.
  - People sitting more than 25 meters apart may not communicate enough.
  - A common area in the workspace, such as a common printer or coffee machine improves communication. People sitting in different buildings do not communicate as much as people in the same building. People sitting in different labs communicate less than people in the same lab. People from different countries may have difficulties, both culturally and with the language. If people reside in different time zones, communication will be more difficult. This is a problem in outsourcing software development.

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

- History of prior use

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

- Local factors

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

- One general factor to be considered in general

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

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

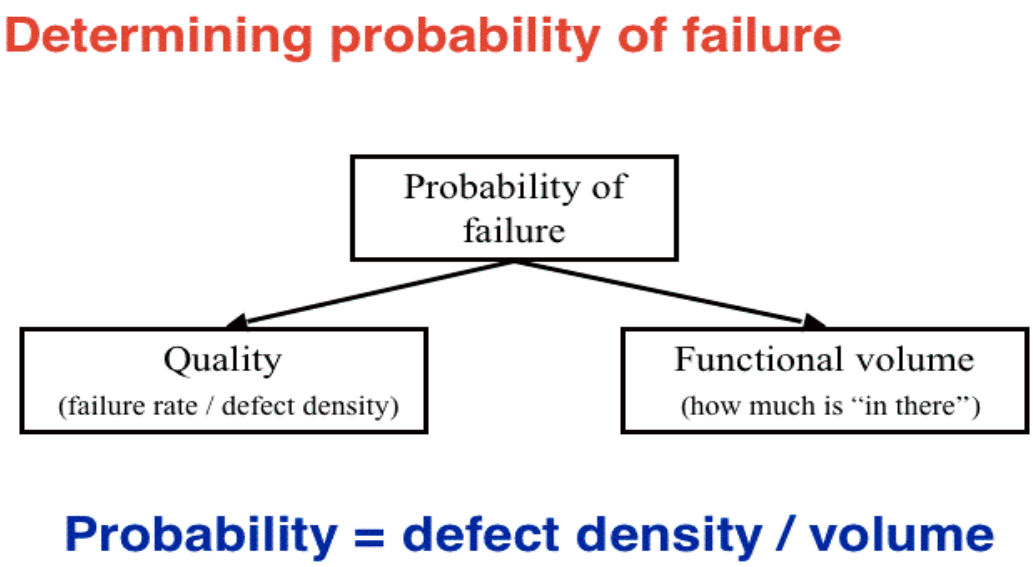


Figure 2: Failure Probability

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

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

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

3.3. How to calculate priority of test areas

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

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

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

The number of points for a factor is then multiplied by its weight. This gives a weighted number of points between 1 and 50. These weighted numbers are then summed up for damage (impact) and for probability of errors, and finally multiplied. As many intuitive mappings from reality for points seem to involve a logarithmic scale, where points follow about a multiplier of 10, the associated risk calculation should ADD the calculated weighted sums for probability and damage. If most factors' points inherently follow a linear scale, the risk calculation should MULTIPLY the probability and damage points. The user of this method should check how they use the method! Testing can then be planned by assigning most tests to the areas with the highest number of points.

An example (functional volume being equal for the different areas):

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

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

Computation is easy, as it can be programmed using a spreadsheet. A more detailed case study is published in (4). A spreadsheet is on http://home.c2i.net/schaefer/testing/riskcalc.hqx (Binhex file, save to disk, decompress, open with Excel)

A word of caution: The assignment of points is intuitive and may be wrong. Thus, the number of points can only be a rough guideline. It should be good enough to distinguish the high-risk areas from the medium and low risk areas. That is its main task. This also means you don't need to be more precise than needed for just this purpose. If more precise test prioritization is necessary, a more quantified approach should be used wherever possible. Especially the possible damage should be used as is, with its absolute values and not a translation to points. An approach is described in (18).

4. Making testing more effective

More effective test means to find more and more important defects in the same amount of time. The strategy to achieve this is to learn from experience and adapt testing.

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

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

Test preparation sets up areas to test, the test cases, test programs, databases and the whole test environment. Especially setting up the test environment can give a lot of trouble and delay. It is generally easy to install the program itself and the correct operating system and database system. Problems often occur with the middleware, i.e. the connection between software running on a client, and software running on different servers. Care should be taken to thoroughly specify all aspects of the test environment, and dry runs should be held, in order to ensure that the test can be run when it is time to do it. In a Y2K project, care was taken to ensure that licenses were in place for machine dates after 1999, and the licenses allowed resetting of the machine date. Another area to focus is that included software was Y2K compliant.

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

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

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

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

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

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

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

5. Making testing cheaper

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

Automation

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

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

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

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

The people factor - Few and good people against many who don’t know

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

Testing requires skill and knowledge. Without application knowledge your testers do not know what to look after. You get shallow test cases which do not find defects. Without knowledge about common errors the testers do not know how to make good test cases. Good test cases, i.e. test cases that have a high probability of finding errors, if there are errors, are also called «destructive test cases». Again, they do not find defects. Without experience in applying test methods people will use a lot of unnecessary time to work out all the details in a test plan.

If testing has to be cheap, the best is to get a few highly experienced specialists to collect the test candidates, and have highly skilled testers to improvise the test instead of working it out on paper. Skilled people will be able to work from a checklist, and pick equivalence classes, boundary values, and destructive combinations by improvisation. Non-skilled people will produce a lot of paper before having an even less destructive test. A method for this is called “exploratory testing”.

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

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

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

6. Cutting testing work

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

Who pays for unit testing?

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

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

What about test entry criteria?

The idea is the same as in contracts with external customers: If the supplier does not meet the contract, the supplier gets no acceptance and no money. Problems occur when there is only one supplier and when there is no tradition in requiring quality. Both conditions are true in software. But entry criteria can be applied if the test group is strong enough. Criteria include many, from the most trivial to advanced. Here is a small collection of what makes the life in testing easier:

- The system delivered to integration or system test is complete
- It has been run through static analysis and defects are fixed
- A code review has been done and defects have been corrected
- Unit testing has been done to the accepted standards (near 100% statement coverage, for example)
- Any required documentation is delivered and is of a certain quality
- The units compile and can be installed without trouble

- The units should have passed some functional test cases (smoke test).
- Really bad units are sorted out and have been subjected to special treatment like extra reviews, reprogramming etc.

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

Less documentation

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

Cutting installation cost - strategies for defect repair

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

When to correct a defect, when not?

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

Rule 1: Repair only important defects!

Rule 2: Change requests and small defects should be assigned to the next release!

Rule 3: Correct defects in groups! Normally only after blocking failures are found.

Rule 4: Use an automated "smoke test" to test any corrections immediately.

7. Strategies for prevention

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

The imperatives are:

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

Measure:

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

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

8. Summary

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

Priority 1: Return the product as fast as possible to the developers,
with a list of as serious deficiencies as possible.

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

References

(1) Joachim Karlsson & Kevin Ryan, “A Cost-Value Approach for Prioritizing Requirements”, IEEE Software, Sept. 1997

(2) James Bach, “Good Enough Quality: Beyond the Buzzword”, IEEE Computer, Aug. 1997, pp. 96-98

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

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

(5) Tagji M. Khoshgoftaar, Edward B. Allan, Robert Halstead, Gary P. Trio, Ronald M. Flass, “Using Process History to Predict Software Quality,” IEEE Computer, April 1998

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

(7) Ytzhak Levendel, “Improving Quality with a Manufacturing Process”, IEEE Software, March 1991.

(8) “When the pursuit of quality destroys value”, by John Favaro, Testing Techniques Newsletter, May-June 1996.

(9) "Quality: How to Make It Pay," Business Week, August 8, 1994

(10) Barry W. Boehm, Software Engineering Economics, Prentice Hall, 1981

(11) Magne Jørgensen, 1994, “Empirical studies of software maintenance”, Thesis for the Dr. Scient. degree, Research Report 188, University of Oslo.

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

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

(14) IEEE Standard 1044, A Standard Classification of Software Anomalies, IEEE Computer Society.

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

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

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

(18) Heinrich Schettler, “Precision Testing: Risikomodell Funktionstest” (in German), to be published.

METHODS & TOOLS is published by **Martinig & Associates,** Rue des Marronniers 25,
CH-1800 Vevey, Switzerland Tel. +41 21 922 13 00 Fax +41 21 921 23 53 www.martinig.ch
Editor: Franco Martinig ISSN 1661-402X
Free subscription on : http://www.methodsandtools.com/forms/submt.php