Test Management is Risk management

Risk Based Testing

Hans Schaefer
Software Test Consulting
Reigstad
N-5281 Valestrandsfossen
NORWAY
Phone +47 56 394880 Fax +47 56 394570
e-mail hans.schaefer@ieee.org
http://home.c2i.net/schaefer/testing.html

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 testing time 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. Chapter 3 of this paper show the idea.

In this paper we are talking about the higher levels of testing: Integration, system and acceptance test. We assume that some basic level of testing of every program (unit testing) has been done by the developers. 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. Doing bad testing you will be the scapegoat for lack of quality. Doing reasonable testing you will be the scapegoat for a 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 a waste of 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.

A good 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. You may require the product to fulfill certain entry criteria before you test. Entry criteria can include completed reviews, 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 second strategy is test prioritization: Test should find the *most important defects* first. Most important means often "in the most important functions". To find possible damage, analyze how every function supports the mission, and checking which functions are critical and which are not. To find the probability of damage, you have to decide where you expect most failures. Finding the worst areas in the product and testing them more will help you find more defects. If you find too many serious problems, management will often be motivated to give you more time and resources for testing. Most of this paper will be about a combination of most important and worst areas priority.

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. Cutting testing work

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 delayed by finding all the defects which could have been found earlier.

As a test manager, you should require higher standards for unit testing!

Use 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 (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.

Less documentation

If a test is designed "by the book", it will take a lot to document. Not all this is needed. 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. Here is how to keep it low.

Cost 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 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. 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.

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.

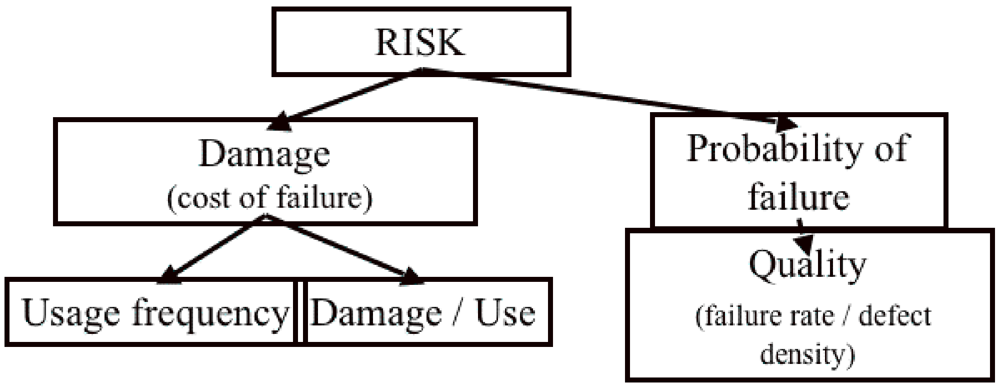

Figure 1: Risk definition and structure

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. A way to lessen the test load is finding the most important functional areas and product properties and concentrate on them. 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 which we want to test more, 1 points is given to areas where we do not see it important to include them in the testing effort at all. (Other publications often use weights 1 through 3). The details of the computation are given later, in table 1.

3.1. Determining damage: What is important?

• Critical areas (cost and consequences of failure)
• Visible areas (where many users, and others, or licensing agencies, are interested in the output)
• Usage frequency

Damage can either be considered on a scale from 1 to 3 or 5, or, better, as absolute values.

3.2. Failure probability: What is (presumably) worst

The worst areas are the ones having most defects, leading to the highest probability for failure. The task is to predict where most defects are located. This is done by analyzing probable defect generators. The most important defect generators and symptoms for defect prone areas are listed below. There exist many more, and you must consider local factors in addition.

• Complex areas
• Changed areas
• Impact of new technology, solutions, methods
• Impact of the number of people involved
• Impact of turnover
• Impact of time pressure
• Areas which needed optimizing
• Areas with many defects before
• Geographical spread
• Areas badly tested before
• Local factors

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!

Table 1: 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

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 in your analysis, you assign a number of points to every product requirement you may want to test (every function, functional area, or quality characteristic. 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 probability points for every factor between 1 and 50. These weighted numbers are then summed up for damage and for probability of errors, and finally multiplied. Testing can then be planned by assigning most test to the areas with the highest number of points.
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[1]. A spreadsheet is on http://home.c2i.net/schaefer/testing/riskcalc.hqx .

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.

4. 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

The long version of this paper is on the Web: http://home.c2i.net/schaefer/testing/risktest.doc

[1] 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! The spreadsheet in the internet uses ADDITION, whereas the example above uses MULTIPLICATION.