What a Tester Should Know, even After Midnight

Hans Schaefer

Software Test Consulting
Chairman, Norwegian Software Testing Qualifications Board
N-5281 Valestrandsfossen, NORWAY
Phone +47 56394880, fax +47 56394570, cell phone +86 13520228154
http://home.c2i.net/schaefer/testing.html
hans.schaefer@ieee.org

A tester can do testing «just as a job». This could be good enough. Alternatively, a tester can invest «the little extra», i.e. do a better job. In this paper, I try to define what this «little extra» means.

1. Testing The Normal Way is Not Enough.

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

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

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

2. Can we do Testing in a Better Way?

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

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

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

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

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

Part of the tester's task is to report incidents. This is not easy. Most literature read by testers just describes issue and defect management, i.e. the life cycle of reporting, prioritizing and handling these. But this is just «the ordinary job». Actually, there is more to it. It can be compared to the task a frustrated user has when calling the supplier help desk: Describing

the problem in such a way that the other side accepts it as important enough to do something about it. It means collecting more information about the trouble, but also to think about how to «sell» the bug report to the responsible person. This is the topic of chapter 6.

Finally, a tester has some rights. We should not just test anything thrown to us over the wall. If information we need is not available, or if the product under test is far too bad, testing it anyway means wasted resources. There should be some entry criteria to testing, some «Tester's Bill of Rights» (Gilb 2003). This is the topic of chapter 7.

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

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

3. The Purpose of Testing

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

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

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

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

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

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

Thus, there are lots of indicators that may lead us to more defects, but we have to use them with care.

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

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

A very basic method for this is looking at the usage profile, and at possible damage. A tester should ask which kind of user would do what how often. How will different users use the

system? How will this vary over time? And very important is not to forget about wrong input and misuse. People have finger trouble, and interfacing systems have bugs. Thus there is not only use with correct inputs, but probably also a lot of use with wrong inputs.

The other risk factor is possible damage. This may be difficult to analyze. A start is at least to ask oneself "what is the worst thing that can happen if this function, feature or request fails?"

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

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

> A pessimist is a better tester!

4. Continuous Learning

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

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

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

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

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

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

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

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

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

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

5. A Critical Mindset

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

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

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

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

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

Don't assume! Ask!

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

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

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

6. Defects

Nobody loves the messenger with bad News!

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

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

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

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

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

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

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

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

However, if intermittent bugs occur, it is a good idea to be able to rerun the same sequence of test cases, maybe even with the same timing, and do more checking than before. James Bach (Bach 2005) has a good guide to investigating such problems:

> Analyze even intermittent problems!
> Log everything you do in testing!
> Log everything you see, and look at more remote details!

Make it possible to rerun your tests, with more logging and analysis tools switched on!

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

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

7. The Tester Has Some Rights

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

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

A tester is NOT a vacuum cleaner!

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

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

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

Testers Bill of Rights:

1.   Testers have the right to sample their process inputs, and reject poor quality work (no entry).
2.   Testers have the right to unambiguous and clear requirements.
3.   Testers have the right to test evolutionarily early as the system increments.

4. Testers have the right to integrate their test specifications into the other technical specifications.
5. Testers have the right to be a party to setting the quality levels they will test to.
6. Testers have the right to adequate resources to do their job professionally.
7. Testers have the right to an even workload, and to have a life.
8. Testers have the right to specify the consequences of products that they have not been allowed to test properly.
9. Testers have the right to review any specifications that might impact their work.
10. Testers have the right to focus on testing of agreed quality products, and to send poor work back to the source.

The last one is the real clue:

Testers should send bad work back to the source!

8. The Late Night Tester's Toolbox

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

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

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

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

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

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

One technique is the basis of most black box testing: Equivalence partitioning. It helps to save test effort, and it can be applied to derive other test techniques. As a tester, you should know it, but also be aware that it has caveats: Black box testing may leave out

important aspects. You should also be aware that combination testing is interesting. Lee Copeland (Copeland 2004) has published a good introduction.

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

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

Test the test environment! Check you test data!

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

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

9. Selected References

- Bach 2005: James Bach. A blog note about possible causes of intermittent bugs: http://blackbox.cs.fit.edu/blog/james/
- Beizer 95: Boris Beizer, Black Box Testing, John Wiley, 1995
- Better Software Magazine, www.bettersoftware.com. www.stickyminds.com Very practical!
- BS7925: British Standard: www.testingstandards.co.uk/bs_7925-1.htm
- Copeland 2004: Lee Copeland, A Practitioner's Guide to Software Test Design, Artech House, 2004.
- Crispin: Lisa Crispin, Tip House, Testing Extreme Programming, Addison-Wesley, 2002, also http://home.att.net/~lisa.crispin/XPTesterBOR.htm
- Gilb 2003: "Testers Rights: What Test should demand from others, and why?"., Keynote at EuroSTAR 2003
- GTB: German Testing Board: www.german-testing-board.info The German Testing Board developed an earlier version of the nowadays-existing ISTQB certification.
- IEEE Standards: See www.ieee.org

- ISEB: Information Systems Examinations Board of British Computer Society. http://www.bcs.org/BCS/Products/Qualifications/ISEB/ has run a certification scheme for software testers since 1999.
- ISTQB: www.istqb.org International Software Testing Qualifications Board. Develops and runs an international software tester certification scheme.
- ISTQB Glossary: www.istqb.org/fileadmin/media/glossary-current.pdf
- Kaner 99: C. Kaner, J. Falk, H. Q. Nguyen, "Testing Computer Software (3rd ed.), John Wiley, 1999.
- Kaner bugadvoc: A presentation about how a tester should report issues. http://www.kaner.com/pdfs/BugAdvocacy.pdf
- Myers 79: Glenford Myers: The Art of Software Testing, John Wiley, 1979.
- Schaefer 2004; Hans Schaefer, "What Software People should have known about Software Testing 10 years ago - What they definitely should know today. Why they still don't know it and don't use it", EuroSTAR 2004