# Things Testers Miss

James Lyndsay, Workroom Productions Ltd.
jdl@workroom-productions.com
London, 2006

## Abstract

Bugs slip into production in spite of the best efforts of designers, coders, and testers.

While testers may not be responsible for the introduction of bugs to the system, they bear some responsibility for the introduction of bugs to the user.

Testing can be adjusted to reduce the number of bugs that pass through to production – without necessarily requiring more resource.

## Terms

**Bug**: something that surprises and bothers the intended user – or one of their friends.

**Problem**: something that causes bugs to be missed by the test team.

## Purpose and scope

As a tester, I've learnt a large chunk of my discipline by doing things wrongly, and working out why. This paper uses real-life case studies to illustrate situations where the test team missed a bug that was later found – and could perhaps have been found while testing. It makes an attempt to fit these case studies into a structure, and gives some positive suggestions about how test teams might reduce the occasion or impact of these misses.

The paper does not attempt to cover ways that the designers and coders might seek to reduce the number and impact of bugs in the untested code. Also, this paper is not intended to be an extended whine about needing more time, more kit, more skill, or more testers.

The paper concentrates on the bug-finding, risk-oriented aspects of testing, while giving less emphasis to the verification-led, value-focussed aspects of testing. This reflects the paper's overall focus on bugs, rather than the relative importance of these very different goals.

## 1. Testers don't make bugs

Testers don't make bugs.

Then again, hardly anyone does. It is more correct perhaps to say that testers can't *avoid* bugs. Testers have to find them.

If you are making something, you can make bugs – sometimes by making a mistake, sometimes by making a decision that turns out to be flawed, sometimes by not noticing that deep within all the ramifications of your actions lies potential for trouble.

Testers are often all too happy to point out these flaws in other people's work, and are even happy to suggest ways that those people could change their work and so avoid introducing similar bugs elsewhere. However, testers are not immune to failure.

Sometimes, testers make mistakes. Testers make flawed decisions. Testers fail to notice that within their actions lie dormant problems. When dealing with the disarmingly simple question 'How did the testers miss ***this***?', the glib misdirection 'Testers don't make bugs' is no answer whatsoever.

To have a bug to log, the test team must **trigger** it *and* **observe** it. Testers miss some bugs because those bugs are never trigged in testing. Testers miss other bugs because, although the bug is triggered, nobody notices.

## 2. Coincidence and test design

For any reasonable system, possible tests far outnumber actual bugs. Bugs are often found during tests that are not designed to look for them – but which trigger them by coincidence.

Conversely, consciously designed tests do not reliably reveal the bugs that surface in production. It is entirely possible (and not uncommon) to have a supposedly exhaustive set of tests that fail to find bugs that appear on the very first day of live use.

Big bugs are often found by coincidence, because the more ubiquitous the bug, or the greater its impact, the easier it is to see – even when using a dumbed-down and mostly-blinded tool. Bugs found by coincidence rather than design also tend to seem bigger because they are more of a surprise.

Test design often concentrates on ways to act on the system to trigger bugs. It is vital to also consider the opportunities for observing bugs – whether triggered by design, or by coincidence.

### 2.1. Observation and diagnosis

Finding a bug by coincidence does not mean that the bug is reproducible. The observed bug may be unrelated to the test activity, and it is hard to judge the effectiveness of individual test techniques when, in the real world, those techniques intermittently find unrelated but important bugs.

In some circumstances, a bug may be logged for diagnosis and decision-making without necessarily being reproducible. However, in other situations, the test team may be required to reproduce the bug before it can be logged. Diagnostic testing is often used to develop a single clear, reproducible test, and sometimes it is possible to look over previous observations to see if an occurrence has already been recorded.

Bugs that are triggered intermittently may not be noticed at all. Even if they are recognised, they may be masked or conflated with a more frequent – or more easily reproducible – failure.

There is an illusion of control with well-designed tests; the illusion that every aspect of the system is under the control of the tester. As systems become increasingly

complex, but present simple interfaces to the user, this illusion becomes less sustainable. If the illusion persists even when testing a system that takes its own decisions in a complex environment (i.e. a game or networked device), the emphasis on test design at the expense of observation will allow bugs to drop through to production.

# 3. Problems of Technique

Of the bugs that arrive in the live, many, but not all, are missed during testing – the tester makes a mistake, uses inappropriate techniques, fails to spot an exploitation that triggers a bug and so on.

## 3.1. Oops – butterfingers

We all make mistakes. By mistake, I mean a moment of sheer incompetence. A mistake is not something that can be put down to a failure of imagination, or a lack of knowledge – it is purely and simply an occasion when we under-perform. Given the same job, two competent people would be unlikely to make the same mistake. Mistakes are common, but are also commonly picked up.

In Case study 01: *What do you mean, Quarterly Billing?*, I describe a mistake of mine; its consequences were significant, and would have been worse had it been recognised later. However, it was so obvious to the competent people around me – and to me, once pointed out – that remedial action was clear and easily prioritised.

The bug was becoming more dangerous as the team started to base more work on it. This happens particularly if the mistake is made early, left unquestioned and used as the basis for decisions. Such mistakes become harder to spot, and there may be resistance to correcting them.

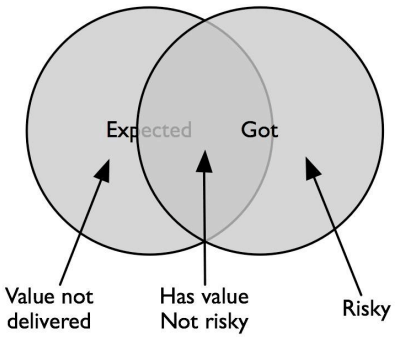
Prevention and detection of mistakes are distinct tasks. In *Using Poka-Yoke Techniques for Early Defect Detection* [1], Robinson describes ways of that testing can help avoid mistakes in software. In my own practice, I have found that diagramming and personal sense-checking followed by peer reviews cheaply weed out most mistakes in testing – and that a swift sanity check with a few people from diverse backgrounds is better for finding mistakes than an in-depth review with experts.

Eliminating mistakes – even in software – does not eliminate bugs. Many bugs can be traced to deeper causes than temporary incompetence.

## 3.2. Trouble with Technique

Some test design techniques (i.e. domain analysis, cause-effect graphing, pair-permutation) set out rules with which to derive tests. This is a great way of digging deep into a particular area, and also provides the stimulus to trigger lots of coincidental bugs.

However, every methodical, limited approach to test design will bypass some family of bugs. To stand a chance of finding these, the underlying limits must be recognised, and in some sense broken by further tests. The more tests share a single underlying approach, the more likely it is that bugs that are avoided by that approach will only be found when the system goes into real use.




Testing focussed on prediction – *Are my expectations fulfilled?* – tends to give good information about value. Tests focused on the deliverable – *Does this system do anything I don't expect?* – tend to give good information about risk. Each will coincidentally give some information about the other, but neither is sufficient by itself. Concentrating testing on the basis of expectation will not tell you as much as you might need to know about risk, while concentrating testing on the deliverable may not tell you whether the system delivers value to its users.

In Case Study 07: *Big Pictures*, testing was driven by a functional breakdown of the system – the performance of the system was not considered testable, leaving an easily correctible bug in a position where it was far more obvious to the end-users than to the test team.

In Case Study 08: *Inspector Gadget*, the bug could be found easily by inspection (which was made much easier by appropriate code layout). The chosen manual, exploratory test approach was unlikely to trigger the bug. Even if the bug was triggered, there was little chance of recognising it – the test had no ready oracle.

In Case Study 09: *Not Random Enough*, brain-engaged testing led me further from discovering the bug. Random testing – with the appropriate constraints on randomness – would have revealed the bug more quickly and more easily. It would have still been up to me to notice that the bug had occurred, however.

BS 7925-2 [2] describes a number of test techniques, Appendix C of the standard deals with test effectiveness as follows: *"Research into the relative effectiveness of test case design and measurement techniques has, so far, produced no definitive results…"*. It cannot be said that one technique is generally better at finding general bugs than another, but it is certain that different test techniques miss different bugs. One way of judging what might be missed is to look at the coverage metrics. BS 7925-2 has a coverage metric for most techniques listed.

In *Software negligence and testing coverage* [3], Kaner outlines over one hundred measures of coverage. While the coverage measurement for a single technique can give you an idea of what might be missed by an individual test approach, the done-ness of the overall test effort should be judged against a range of coverage metrics. To quote Appendix C again; *"There is no requirement to choose corresponding test case design and test measurement techniques."*., Testing may miss that which is systematically missed from coverage.

Using diverse test techniques can help avoid this trap – see *Lessons Learned in Software Testing* [4] Lesson 283 for more.

### 3.3. Randomness

Random testing involves giving random input to a system. The randomness is usually constrained in some sense; the constraint is either characteristic of the test subject or environment, or an element of the test design.

Random testing triggers bugs by coincidence, rather than by explicit design. Where automated, random testing is cheap and speedy, but the difficulty of observation can mean that only the biggest bugs are observed, and that the results that are hard to diagnose or simplify. The paper *Fuzz Testing of Application Reliability* [5] describes a random test approach that revealed crashing or hanging bugs in many otherwise resilient test subjects.

In Case Study 09: *Not Random Enough*, I describe a test approach that would have benefited from more randomness. Many brain-engaged testers find it hard to be usefully random – so the technique often needs some kind of tool support. Randomness can be applied in many ways – the tester may want to consider which of the available ways will be simplest to execute and observe before progressing to more complex designs for random tests.

### 3.4. Exploitation

In Case Study 06: *Compounding Errors*, I describe a situation where a bug was recognised, but its importance was not. Exploitation puts an observed bug out of the context of its initial discovery, and into the context of how it could affect the real world.

Exploitation relies on a testers understanding of that real world, and often requires detailed knowledge of the underlying technologies, supporting systems and business context. The impact of bugs in Case Study 06 and in Case Study 03: *Too Many Errors* could potentially have been spotted during the design phase by considering potential failures and exploitation.

It can be productive for testers to work with designers or coders to identify such potential bugs before they are made real in implementation. It can also be possible to take advantage of the business and technical knowledge of those same designers and coders when exploiting a small bug. See Whittaker's *How to Break Software* [6] for many approaches to exploitation.

A system is often most exploitable at its limits, or when part of it is already failing. Bugs found at the limits may be discounted as 'would never happen in normal use', but these limits may be the only points in the test lab where a common-in-live bug may be triggered. See *A Positive View of Negative Testing* [7] for more.

### 3.5. Emergent behaviours

In Case Study 03: *Too Many Errors*, Case Study 04: *Too many files*, Case Study 05: *Conflict of interest* and Case Study 06: *Compounding Errors*, I describe situations where the delivered system displayed characteristics that were neither designed, nor avoided in design. The bugs were not due to mistakes, but to unexpected interactions.

Many important bugs turn up out of the blue. With increased analysis and awareness, they might have been avoided – but few projects are in a situation where enough time and skill can be applied to feel genuinely confident that they have been eliminated. Some behaviours may emerge only when an existing system comes into contact with a novel situation, interacts with a new system, or receives the close attentions of a skilled hacker – or exploitative tester.

Few tests are designed with emergent behaviours in mind, so coincidental triggering and observation is vital. A test approach which relies on scripted testing and narrowly-automated observation is likely to miss emergent behaviours.

## 4. Facilities and Procedures

Some bugs are missed because of problems with the facilities and procedures used by the test team.

### 4.1. No chance

Some bugs stand very little chance of being seen before live operation, because the opportunities to trigger or observe them are unlikely – or impossible – in the test environment.

In Case Study 04: *Too many files*, I describe a lab that had **no chance of triggering** the bug described. The lab was using an efficient and convenient approach to test data generation, but this approach avoided the bug. The bug was unanticipated – and the cost of simulating the production environment was too great to be justified.

In Case Study 07: *Big Pictures*, I describe a lab that had **no chance of observing** the bug with their chosen test approach. It is worth noting that they would have easily seen the bug with analysis, or use of a simple tool.

In Case Study 02: *Too many boxes*, I describe a lab that was **unlikely to trigger or observe** the bug simply by observing operation of the kit during normal testing. The live environment had more than 100 times the number of installations – the test team would have had to be lucky to observe the bug.

Many test labs are just not big enough to coincidentally observe a bug that might be regular in production. Software that is sold in bulk can have a user base many orders of magnitude larger than the potential test lab – think of games consoles and mobile phones.

**Illustrative Calculation:**

- Imagine a team that is actively testing for 30 days.
- The kit they're testing has a chance of failure of about 1% a day. At three times a year, that's not a terribly unusual problem, but the chance of not seeing the bug on any given day is 99%. The chance of not seeing the bug in two given days is 99%*99%. In three, it's 99%*99%*99% and so on.
- Chance of seeing no failures at all in 30 days = (99%)^30 = 74%. That's 100-74% = a 26% chance of seeing the bug at least once.
- With 30 days testing, they would have just better than a 1 in 4 chance of observing a failure.
- With a test lab that contained four sets of kit, rather than one, that chance rises to about 2 in 3.

To shine the light of the real world on this hypothetical example for a moment, it is worth wondering how long 30 days of active testing might take in practice. Also consider the changes in code, data and configuration that could happen over that period – and how that might affect the credibility or diagnosis of a bug observed just a couple of times.

### 4.2. Data

Data corruption is at the heart of many frustrating bugs. The effects of corruption can be both unpredictable and distant from the point where corruption was introduced. This disconnect can make the original cause next-to-impossible to diagnose, and a single persistent source of corruption can be recognised as general unreliability throughout the system.

Corruption can be picked up close to the point where it occurs by taking some kind of data snapshot before and after tests, and using a basic comparison tool to highlight differences. This approach can help to isolate the source, and to make the bugs more easily reproducible and recognised. With simple tools, the approach can be used with flat files, database dumps, configuration data, windows registries, installed software libraries or versions. See *The Importance of Data in Functional Testing* [8] for more.

Bugs are not restricted to code. Data can introduce bugs, particularly in systems that rely on configuration data. In Case Study 10: *It's the Data, Stupid!*, I describe a critical bug that existed only in third-party data. The data was not part of the test subject – but the bug could have been found in testing by exploring the data before it was promoted to the live environment.

### 4.3. Signal vs noise

Some organisations that have a lot of bugs related to the testing and software build processes also let a lot of bugs through to production.

Genuine bugs can be masked by test bugs – that is, they cannot be triggered or observed because a test-related bug happens first. If the bugs in the test system are well understood, and the test aimed at a specific bug, it may be possible to avoid test-related bugs, while still triggering the bug or validating the functionality. However, the test lab's ability to trigger and observe coincidental bugs will be reduced. A better solution is, of course, to reduce the incidence of test-related bugs.

Where an automated test is over-sensitive to test-related problems (or is out of sync with the system to be tested), testers may decide to allow the test to pass by reducing the scope of the observation, or turning it off entirely. It is not uncommon for these temporary modifications to become permanent, so allowing later bugs to slip through unobserved. It may be possible to avoid this creeping problem by using a time-limited version of the test, but a better long-term solution is, of course, to build more robust/maintainable test suites.

Test-related bugs affect the credibility of genuine bugs found by the test team – especially if some test-related bugs have been logged as code bugs and returned by the coding teams. This can increase the resistance to genuine bugs that are intermittent, dubious, or even hard to fix. Once again, the best solution is to sort out the test lab, but if this cannot be addressed, it may be useful to allow a bug to have multiple priorities/severities, each set by a different interest group.

### 4.4. Rules hidden in procedures

Underlying rules in test procedures can have a similar effect to monotonic test design techniques: Testing that is conducted systematically can systematically miss groups of related bugs.

Some of those underlying rules are, however, foundations of commonly-recommended good practice/ For example:

- Resetting the environment after every test

*good for clean results, but will miss bugs that leave the system so corrupt at the end of the action that the next action is to crash*

- Varying only one thing at a time

*good for diagnosis, but will miss bugs related to interactions between changing variables.*

- Following user profiles

*good for 'realistic' tests that show value, but will miss bugs that are obvious to users who don't fit expectations*

- Using well-designed test data

*good for clean results and flexible tests, but will avoid bugs that show up in large volumes of dirty data.*

It would be foolish to avoid all the benefits of these helpful practices by throwing them out entirely. Nevertheless, it is worth looking out for these and others in one's own testing, and to consider what might be missed.

### 4.5. Broken requirements

In Case Study 05: *Conflict of interest*, I describe a situation where requirements were known, but in conflict and incomplete. Testing that is based solely on requirements may happily pass conflicting requirements individually, with the results of their conflict unresolved and observable to all in live.

Missing requirements, like missing tests in test-driven-design, can mean that a system looks good when assessed point-by-point, but is clearly incomplete when looked at holistically.

The problems can be addressed by driving test skills up-stream, to be involved in requirements gathering and analysis. However, the process of design is a process of change – as more design is made real, understanding of what is required will change. Gathering requirements involves an extraordinary exercise of the imagination by all concerned; testers can facilitate and enhance that exercise.

## 5. Problems to be addressed at project level

Some things are missed because of what can be seen as 'project-level' problems. The test group did a great job – remedial action lies outside the group, and someone else must take action to avoid the problem in future.

Typical thoughts that occur in these situations might be:

*We'd have found this bug – if we'd been able to spend more time testing.*

or

*We found and reported the bug, but it wasn't taken seriously.*

These bugs beg to be used as examples to help the overall organisation understand where its values lie. Should more time be allotted to testing? Should more attention be paid? If genuine, such bugs can be visceral evidence to decision makers, and can directly affect not only future plans, but also the credibility of the team itself.

Nevertheless, people with the power to influence testing from outside the group will often take an alternate viewpoint. They will ask pointed questions. It is good to be prepared for these questions – and not simply because it makes answering them easier.

There is a chance that responsibility and opportunity for addressing these problems does not, after all, lie with someone else. Asking these same questions within the team allows the testers to make pre-emptive steps and take ownership of necessary elements of the problem. Not only is this an effective route towards a fix, it strengthens the position of the team in negotiation. It is possible that changes outside the test team will not happen unless it is clear that an aligned effort is taking place with the group.

### 5.1. We needed more stuff

Some test teams feel that with more time – or more resource – they would have found any bug that dropped through to production. Rather than present a case study, I encourage readers to write one based on their own experience, and consider their reasoned responses.

Where deadlines were flexible, questions might be:

- If more time was needed, was more time requested?
- At what point was it recognised that more time might be needed?
- Why wasn't the need for more time recognised in planning?
- What could be done to improve the plan and estimates?

If available time was fixed, questions may run more along the lines of:

- What did you do that you could have avoided?
- Could you have gained time from elsewhere – by starting earlier, or working longer hours?
- Could this be a prioritisation issue?

### 5.2. We weren't taken seriously

Case Study 06: *Compounding errors* provides an illustration of this situation. All parties were aware of one of the key factors that led to the system's eventual failure. Situations like this present a fine opportunity for the organisation to learn and to adjust its behaviour.

Questions that might be asked of the test group include:

- Did the test team recognise the importance of the failure, or its potential to cause harm?
- Did they communicate the information they had?
- Was there enough time to fix the bug – and if not, what would have been required to find the bug earlier?

However, by ignoring the bug first time round, those outside the test team may have already indicated a resistance to learning. It may be hard to gain a rational response – or, indeed, any useful response at all. Some project-level problems either cannot be fixed, or will not be fixed. You might hear something along the following lines:

*It's not a bug – it's a mis-used characteristic of the system. We need to educate our users.*

or

*We can't do anything else within this technology – there is no other solution.*

### 5.3. The Big Squeeze

Testing may have been shortened, or required to take a late change. The reduced opportunity to find the bug can mandate focussed testing, which is more likely to miss a coincidental bug that larger scale, more diffuse testing might have picked up.

This situation can be the trigger to initiate a sledgehammer response: to stop late changes, or to try mandatory automated regression testing in an sttempt to introduce faster, wider testing. To be effective, such solutions need to be all-encompassing and so can introduce on the one had unacceptable friction, and on the other hand, impossible engineering.

A better solution might be to use a different technique; greater inspection of code related to late changes, or an understanding and acceptance of the gambles involved in risky behaviour. If your environment regularly ships last-minute bugs to the test team, you may have few alternatives but to ship those bugs to the customer.

## 6. Issues of management

### 6.1. Avoid monotony

If concentrating on one approach misses bugs outside that approach, then diversity is key. Spending time thinking of an alternative and testing within it may rapidly reveal problems in testing that otherwise would be rapidly revealed in production.

Here are some scales, and some monotonic approaches, that may be worth considering. You should be able to think of more. How does your testing stack up?

- action : observation
- design : coincidence
- all-scripted : all exploratory
- automated : manual
- reductionist : holistic
- functional : non-functional
- up-stream : down-stream
- test data : production data
- rationality : randomness
- Driven only by risk
- Driven only by requirements

- Single-user
- Available resources
- Co-operative systems cooperating
- Clean data
- Clean environment

### 6.2. Phases

Different parts of testing have different opportunities to find bugs. While it is commonly agreed that the earlier a bug is found, the cheaper it is to fix, it may not be possible to find all bugs early.

For instance, individual components may show reasonable throughput, but when integrated, bottlenecks emerge. Integrating a system may reveal a need for special cases and new error-handling. Emergent behaviours do not tend to show up at unit test. However, broader observation may reveal side-effects, and considerations of exploitation and attacking behaviour can reveal bugs that are hidden – but fixable – in earlier phases.

Bugs that are found too late tend to be intermittent, as their immediate causes and effects are indirectly related to the input and output of later test phases. Their depth may mean that they have wide-ranging consequences, and if they are fundamental to the system, there may be a cost related to test maintenance or regression testing.

### 6.3. Metrics

The metric 'Defect Detection Percentage' can be used to summarise overall figures of bugs that have been missed in testing. While no substitute for detailed study of individual cases, it may reveal collective information where individual enquiry does not.

DDP has been described by Graham et al [9] and Craig [10]. It is a measure of what was found, as a proportion of what could have been found. Typically, it is calculated as the bug count within a phase as a proportion of bugs found in that phase and beyond. A DDP of 100% would imply that no bugs had been found after the completion of a particular test phase. This could be taken to show that testing had been highly effective. Clearly, this interpretation relies on circumstance; if no-one has reported bugs after the completion of testing, it may be that the system is not in use.

As with any metric, DDP comes with caveats. The method as described pays no attention to severity, and is available only after the event. It does not allow comparison between test techniques, only phases. A level of DDP might be acceptable, based on characteristics of test environments; usability testing may not be an appropriate place to pick up some performance testing bugs. As bugs are found, DDP inevitably drops, and what was judged good at one point might be judged poorly with hindsight. Basing key decisions on DDP can politicise understanding of what is a defect, and which phase it 'belongs' to. Statistics from test phases that have the opportunity to find many bugs by coincidence may give inappropriate value to the non-coincidental techniques in use.

## 7. Conclusion

This paper will give small comfort to teams that have no testers, or teams that have bad testers. However, good testers are engaged in a continual learning process, and will take the opportunity to learn from their problems. In the appendix, I have included the questions that helped me develop my case studies; they may be of use in developing your own.

It can be informative to study bugs that have been caught by others, and to ask yourself if your team might have caught them. Check back issues of *STQE/Better Software*'s semi-regular feature *Bug Report* (for an example, see Doug Hoffman's fine article *Exhausting Your Test Options* [11]). Look through bug taxonomies from Kaner et al [12], Beizer & Vinter [13], Vijayaraghavan [14]. Ask yourself: would the tests you design have caught the bugs? Would you have had the opportunity to trigger and to observe them by coincidence? Would you have recognised the bug if you had?

As testers, we should be able to ask such pointed questions of ourselves, and of our teams, and to use them as the basis of our own process improvements.

### 7.1. Bullets, please

- Big bugs are often found by coincidence
- Recognise monotony and introduce diversity
- Learn from your mistakes

## 8. Acknowledgements

I would like to acknowledge the contributions of the many people and teams who have been part of my own ongoing learning process. Particular credit should go to those individuals who were happy to let me use my case studies – you shall remain forever anonymous! Specific thanks are due to James Bach for hitting me with the exercise in *Case Study 09: Not Random Enough*.

## 9. References

[1] Robinson, H, *Using Poka-Yoke Techniques for Early Defect Detection* , STAR'97, http://www.geocities.com/harry_robinson_testing/pokasoft.htm

Also see Grout's Poka-Yoke Page, http://csob.berry.edu/faculty/jgrout/pokayoke.shtml

[2] Reid, S et al, *BS 7925-2:1998 Software component testing* http://bsonline.techindex.co.uk/ or, more cheaply in draft from: http://www.testingstandards.co.uk/Component%20Testing.pdf

[3] Kaner, C, *Software negligence and testing coverage*, (Keynote address) Software Testing, Analysis & Review Conference (STAR), Orlando, FL, p. 313, May 16, 1996. http://www.kaner.com/pdfs/negligence_and_testing_coverage.pdf

[4] Kaner, Bach, Pettichord, *Lessons Learned in Software Testing*, (Wiley, 2002), Lesson 283 "Use diverse half-measures"

[5] Miller, Fredricksen, So, *Fuzz Testing of Application Reliability*, (University of Wisconsin, 1989) http://www.cs.wisc.edu/~bart/fuzz/fuzz.html

[6] Whitaker, J, *How to Break Software*, (Addison-Wesley, 2003)

[7] Lyndsay, J, *A Positive View of Negative Testing*, (Keynote address, STAREast, 2003) http://www.workroom-productions.com/papers.html

[8] Lyndsay, J, *The Importance of Data in Functional Testing*, (Quality Week 2001) http://www.workroom-productions.com/papers.html

[9] How to measure test effectiveness using DDP (Defect Detection Percentage), http://www.grove.co.uk/pdf_Files/DDP_Tutorial.pdf

[10] Systematic Software Testing by Rick D Craig, Stefan P Jaskiel p277

[11] Hoffman, D, *Exhausting Your Test Options*, (STQE magazine, Jul/Aug 2003 (Vol. 5, Issue 4)), p10-11 http://softwarequalitymethods.com/Papers/Exhaust%20Options.pdf

[12] Kaner, Falk & Nguyen, *Testing Computer Software*, 2nd Edition (Thompson, 1993)

[13] Beizer, Vinter *Bug Taxonomy and Statistics* (2001) http://inet.uni2.dk/~vinter/bugtaxst.doc

[14] Giri Vijayaraghavan, *A Taxonomy of E-Commerce Risks and Failures*. (Master's Thesis, Florida Institute of Technology 2002). http://www.testingeducation.org/a/tecrf.pdf

# 10. Appendices

## 10.1. Case Studies

The following case studies are drawn from my own time in testing: I've been involved in doing, managing or assessing each of these tests. I've tried to keep these case studies as accurate as possible, drawing on my notes and bug reports if possible. Any inaccuracies are my own responsibility.

I've tried hard to avoid including identifying details. If I was under a non-disclosure agreement, I've approached the people or organisations involved before including the case studies here. Some minor details have been changed to protect anonymity.

**Case Study 01: What do you mean, Quarterly Billing?**

*Discovery:* I was presenting intermediate test results to the client. The client said "That's great for our monthly customers – are you taking the same approach with our quarterly billed customers?". We had *no* tests for quarterly-billed customers. The immediate effect was that we had to write and execute some new test cases, re-design some existing tests, and take an operationally different approach to cycling the billing system.

*Testing:* We didn't spot this bug because we'd gone straight from the system specification to scripted tests – and had missed the one place where this was stated explicitly. We were also relying on a mock-up of customer data, which had no quarterly-billed customers. We hadn't presented the ideas of our tests to the business/analysts/designers – and hadn't absorbed requirements, done a sense-check or peer review.

*Resulting Change:* If I have to derive a "complete" set of tests from a document, particularly if the tests involve permutations, I now consider taking an intermediary step of making tables to describe an aspect of the system. This abstraction is easier to communicate, to update and to use in test design (and may ultimately become part of the primary documentation). In this case, I'd have made a table of all the things that characterised accounts in a way that mattered to the business (as columns), and the different ways those characteristics could appear (values in the columns). If I did this reasonably, I would have a column labelled 'billing period', with just the one value 'monthly'. I hope that I would question my assumptions at this point!

**Case Study 02: Too many boxes**

*Discovery:* A software+hardware system seemed reliable in the lab. However, it was unreliable in live operation. With hundreds of separate installs, it seemed that every day a new installation failed, and would need to have an engineer dispatched to fix it. The underlying issue was a hardware failure, and the fix was a manual re-boot; but the hardware failure prevented the system from re-booting automatically.

*Testing:* The test lab had two working systems. To have a chance of seeing this bug, the systems would have had to be left running for many tens of days. Although each system automatically re-booted each day

to avoid encountering long-run-time bugs, this fault was unexpected – indeed, it was not recognised as a system fault until it had happened many times for the customer.

*Resulting Change:* Both supplier and acquirer recognised that these faults would be hard to see in a test lab without unavailable investment in time and kit. They introduced a phased pilot, looking to extrapolate live reliability from measured failures over tens of days in dozens of installations. They also used more internal logging for diagnosis, and paid more attention to bugs found in live.

**Case Study 03: Too many errors**

*Discovery:* Found in live operation, after an unrelated failure caused some of the input data to become invalid. Each invalid record was put into a 'suspense' area for later processing – but the numbers involved went from 10s a day to 100,000s an hour. The suspense system rapidly consumed its available disk space, and manual tools and processes designed to resolve individual records were swamped. Without suspense handling, the system refused to process more input. Another queue started to build, SLAs fell like dominoes, while the big kit stood idle…

*Testing:* Bug was missed because no testing was done on performance of the error-handling system. Performance testing was itself restricted by kit availability – but limited worst-case analysis did not cover this situation and flag it up as a potential problem.

*Resulting Change:* Paid more attention to error-handling systems and worst-case scenarios. Started to draw diagrams of 'load pressure' – the external stresses on a system and the ways it dealt with those stresses internally.

**Case Study 04: Too many files**

*Discovery:* In live, maximum throughput seemed to be rather less than expected, causing unacceptable delays in processing time. We observed that the system would process a large queue more slowly than a small queue – and this was related to the number of files in the queue, rather than the total amount of information to process. The processing system made less use of its CPU at times of high load than at times of medium load, as it was waiting for a much smaller delivery system to hunt through a large list of accumulated files for the 'next' file.

*Testing:* We did not observe this in testing because the files that were created were named in a way that meant that the 'next' file was also the first file in the directory list, so the delivery system spent less time looking for it.

*Resulting Change:* We modified our test data generation to give input files more realistic names. I became more sceptical of predicting live performance based on test system bottlenecks, and paid more attention to modelling the system.

**Case Study 05: Conflict of interests**

*Discovery:* When the system went live, it took hours to initialise. This would have been acceptable, but the system needed to be restarted every day to take in new configuration data. The slow startup was driven by a design decision to get everything into memory, and was based on a need for fast performance while running. The daily restart was driven by a need to avoid time taken by a record-to-record choice of which calendar-driven set of configuration data to apply – again, based on a need for fast performance.

*Testing:* Initialisation time was ignored because the test system was small, and the initialisation was of test data. Any bugs in production would be 'tuned out'. The conflict in design decisions (or requirements, depending on viewpoint) was not seen by testing because the design was opaque to the test team and to the client. The opacity did not result in independence, but obscurity.

*Resulting Change:* I try to understand how the system's operators expect to use or control it, and how its designers expect it to work – and to spot and question differences before go-live.

**Case Study 06: Compounding errors**

*Discovery:* On the first day of live use, the previously working and piloted system was unresponsive and eventually crashed. The central server had many satellite systems – on the day before go-live, they'd been loaded with live information. This was partly corrupt; when the clients were turned on, they asked the server to resend the information. This took minutes per client (a known characteristic) – but worse, did not scale well. The satellite systems waited patiently for hours to receive the information – their users were impatiently locked out. The users (as might be expected) rebooted their satellite systems…

*Testing:* The data corruption was related to record size. It was not recognised because no test existed for it in unit testing, and system testing relied on unit testing for 'edge-case' tests. Being a test-driven development, no code existed to deal with large data – so the tests executed all the code, and the full test suite passed. The scalability of re-load had not been tested, as the scenario was considered to be unlikely.

*Resulting Change:* The situation provided timely reinforcement for me that TDD produces systems that give you a level of confidence that things are working – but with very little information about risk or emergent behaviours. I started to think of extrapolation of interacting characteristics and risks as an analysis problem.

**Case Study 07: Big Pictures**

*Discovery:* The website took over a minute to load. The lion's share of this time was a large background picture. This was noticed during beta testing, after many weeks of functional and usability testing.

*Testing:* The in-house test team had broadband access, and did not notice the loadtime. They concentrated on functional testing. Although they had access to static test tools that would have flagged this (and other errors), they did not use them as most of the pages were dynamically generated based on a session key, and were not well suited to the available tools.

*Resulting Change:* Improved use of static tools and pre-beta user observation.

**Case Study 08: Inspector Gadget**

*Discovery:* While fixing an observed bug in some of my own code, I found a coding error that I had not noticed during testing.

*Testing:* The coding error was directly observable in only 2 out of >600 permutations – and moreover, differentiating correct from incorrect without an oracle was not trivial. It was far simpler to see the bug during inspection.

*Resulting Change:* I inspect my code with greater precision. In particular, where individual lines do similar things, I try to construct and format the code to allow me to directly compare the differences between lines. I also use printouts.

**Case Study 09: Not random enough**

*Discovery:* This was an exercise given by one tester to another – the target bug was described, but I had to find evidence for it myself. The target bug is to do with validation; the system says your input is **in**valid for some **valid** inputs. The input is four sets of numbers, and manual validation is obvious. It took a while to see the bug, but there's a simple technique that would have seen it faster. Why didn't I use this technique?

*Testing:* I've got my test results. Looking back over them, most tests have a rationale – either progress down a particular path, or a new path entirely. I try lots of good ideas – and still don't see the bug. I'm looking for the system's patterns of behaviour – but what I don't see are my own patterns – and how they're skewing my test.

*Resulting Change:* All bugs are obvious, if you know where to look. It turns out this one's obvious if you don't know where to look. Choosing *random* behaviour, and picking specifically the *right* way to be random was key to this one. I don't want to give more away…

**Case Study 10: It's the data, stupid**

*Discovery:* Customer bills were related to distance between known postcodes, calculated on the latitude/longitude of the postcodes. Certain customers were always billed at the maximum amount. On investigation, it became clear that this was because their correct postcode was incorrectly linked to a location in the middle of the North Sea.

*Testing:* Functionally-focussed testing had verified the distance calculation and related billing – but no attention had been paid to the quality of the third-party data.

*Resulting Change:* Where data is to be used by a live system, I consider running sense-checks on the data before it is used. It can be important to put such data under configuration control, not only to restrict unauthorised change, but also to spot changes.

While developing these case studies, I tried to keep the following questions in mind. I hope they are useful when thinking about your own situations.

Discovery: *What was the bug? Had it already had an impact on the business or project? When was it found compared to when you had the opportunity to find it? How was it recognised – was the underlying fault found itself, was a directly related activity noticed, or did it take some diagnosis of an event or an emergent property before the flaw was acknowledged?*

Testing: *Why do you think the bug was not spotted during your own testing? How could it have been seen? Do these observations fit neatly into problems with planning / execution / observation / analysis? With hindsight, what activity might you have chosen to leave out to get the time to see the bug? What information would you have been able to collect that might have helped you recognise / make that decision?*

Resulting change: *Did missing this bug tangibly change the way you approached testing? How did the missed bug affect the way you tested? How did it affect your management of the task and the procedures you may have been working within? Was your organisation affected by the fact that the bug was missed (as distinct from any effect the bug itself may have had)? Were these changes generally for the good? Were there unanticipated consequences of any changes?*

*What was the important context?*

*Don't spend time justifying doing what you did.*

*Don't simply say that you would have done a different job if you'd had more time / kit / people.*

I'd be delighted to learn from your case studies, and to post them for others to learn from: please send them to jdl@wokroom-productions.com.

## 10.2. Rights