‘Good’ Organisational Reasons for ‘Bad’ Software Testing: An Ethnographic Study of Testing in a Small Software Company

David Martin[1], John Rooksby[1], Mark Rouncefield[1], Ian Sommerville[2]
[1]*Computing Department, Lancaster University, UK*
[2]*School of Computer Science, University of St Andrews, UK*
[d.b.martin, j.rooksby, m.rouncefield]@lancaster.ac.uk, ifs@dcs.st-and.ac.uk

Abstract

In this paper we report on an ethnographic study of a small software house to discuss the practical work of software testing. Through use of two rich descriptions, we discuss that ‘rigour’ in systems integration testing necessarily has to be organisationally defined. Getting requirements ‘right’, defining ‘good’ test scenarios and ensuring ‘proper’ test coverage are activities that need to be pragmatically achieved taking account of organisational realities and constraints such as: the dynamics of customer relationships; using limited effort in an effective way; timing software releases; and creating a market. We discuss how these organisational realities shape (1) requirements testing; (2) test coverage; (3) test automation; and (4) test scenario design.

1. Introduction

Despite advances in formal and automated fault discovery and their increasing adoption in industry, it appears that testing, whereby software is ‘shown to be good enough’ will continue as the principal approach for software verification and validation. The strengths and limitations of testing are well known (e.g. [19]) and there is healthy debate over automation (e.g. [3][20]). Case studies (eg. [15][24]) have proved valuable, and following in this programme of ‘empirical studies of testing’ (see [12]) we seek to better describe the practical issues in testing for a small software company.

Best practice in testing has been largely uncontroversial, it being to adopt a phase based approach (see e.g. [19][10][13]). The earlier phases in these models have increasingly been automated (e.g. unit testing [2]), whereas innovations focused on the latter stages have been more human centric (for example risk based testing [1]). Agile methods, such as extreme programming (XP), disrupt such models with test driven development and a rejection of any testing that cannot be fully automated [8]. The agile approach has been successful [24] but there remains a lack of empirical evidence about such testing [18], and we are concerned as to whether it solves or merely displaces certain issues. Our experience is also that many companies who have adopted XP practices, do not, in fact, automate all tests.

Alongside the ‘best practice’ approaches there continue to be more pragmatic guides to testing. For example Whittaker [26] argues that “there is enough on testing theory” and looks at “how good testers actually do software testing”. Kaner [14] provides wider “lessons” based upon his experiences in testing. From such guides it seems that drawing and learning from ‘experience’ is somehow as important as following a rational approach to testing. The empirical study in this paper confirms what Whittaker calls for elsewhere [25]: for theory based and practice based approaches to communicate and converge.

In this paper we discuss the pragmatics of software testing for a small software company. The company, which we shall refer to as W1REsys, follow a programme of automated unit testing and a semi-automated programme of integration and acceptance testing. We focus on systems integration and acceptance testing and find the notion of ‘rigorous’ testing is defined organisationally rather than in accordance with some technical criteria. We discuss why it is important for software engineering researchers to understand that testing is a socio-technical rather than a technical process and that, for product companies there will inevitably be ambiguity related to integration and acceptance testing.

2. Empirical studies of software testing

There is a recognized need for empirical studies of software engineering, including ethnographic studies. One of the strengths of ethnography is the ability to take a broad focus on work rather than on a particular method or technology. Studies of work in testing are rare. Evans [10] gives a “fictional account of a real testing environment … based on several actual project situations in which the planning was poorly done” (p187). Evan’s example shows that “without planning, structure and order the testing of a system and software components has little chance for success”. This fictional example successfully demonstrates how personal, organisational and procedural problems spiral and blend within the overall problem of software testing and emphasises that unique features of the situation affect the ways in which successful testing can be achieved (the example shows how a program support librarian was key to a success because “she was hard as nails.”)

There is a need for real world data, but this too, as Collins [7] discusses for the transportation industries, can fail to distinguish between ‘trying’ (extensively testing the various limits of a technology) and ‘showing’ (providing a demonstration of the technology passing particular tests for a particular audience). Many descriptions in Software Engineering research of the application of testing and verification methods to real world problems, whilst welcome, should be considered more-so as demonstrations than as reports of how testing is done in practice. There is a need for empirically gathered data about testing that has no vested interest in demonstrating the superiority of a particular method or technology. Such studies are rare but include: Martin et al’s [17] discussion of the integration testing of two healthcare technologies; Runeson’s [22] investigation into unit testing practices; Eisenstadt’s [9] collection of war stories about ‘hairy’ bugs; Knuth’s [16] discussion of using a log-book to document errors; and Stringfellow and York’s [23] description of component testing of a radar control system. Empirical studies of “evaluation” can also be helpful. For example Blythin et al’s [4] study of what counted as ‘success’ and ‘failure’ for two groupware systems.

3. An ethnography of software testing

This paper is based upon an ethnographic study of software developers in a small software company that develops a software product for business customers. The study employed observational methods and in-situ interviews to view, capture and understand work as it happened via note taking, video, photographic and audio recordings. A total of 30 days fieldwork were undertaken in a period between July 2005 and April 2006. Our approach to analysis has been ethnomethodological (see Button and Sharrock [5] for a similar approach in software engineering). This approach is to focus without prior hypotheses on understanding how plans and procedures are implemented in practice, how participants coordinate their work, how they reason about their work and organise their activities as a recognisable social accomplishment. Here we are also interested in how the developers reason about customers, the market, requirements, the developing design and testing as they carry out their work.

3.1. Working practice at W1REsys

The company produces a ‘write once, run everywhere’ (w1re) development environment for end users to develop applications (in XML) to run on mobile devices such as mobile phones and pocket PCs. The company has seven full-time employees, four of whom are programmers. The programmers at the study site use practices from XP such as story cards, an on-site customer and frequent releases. However, they have not adopted ‘pure’ XP in that they do not always practice pair programming or automate all testing.

‘W1REsys’ was set up to take advantage of a niche in the market for application development for mobile devices. Due to the differences in mobile devices, the same application has to be programmed and tailored in different ways for e.g. pocket PC’s and various mobile phones. W1REsys’s niche is to produce a development environment with an integrated translation ‘engine’. This allows end users to script applications for various mobile devices in XML.

W1REsys have several established customers in industries such as vehicle repair assistance, couriering and supply delivery, but they are continually looking to enhance the product, maintain customers on license, and expand their market. They have produced a ‘generic’ application for a market area, rather than a specific system for a specific customer or even a specific sector, and are in the position of always seeking to enhance the application and expand their market.

3.1.1. Handling requirements. When they are producing new requirements for ‘the next iteration’ W1REsys are engaged in a process whereby they attempt to work out ‘what would be best to do next?’ They do have long-term aspirations and goals but essentially define their development to take account of immediate circumstances such as the possibility of new customers, new requirements from existing customers, etc. We believe that this fluid approach to product strategy is likely to be common to many small software product companies where the principal priority is survival.

What defines the outcome of this process is influenced by the ideas of the team, particularly the more senior members of the company in charge of sales, marketing, training and strategy. They are the ones who provide the programming team with new requirements to investigate and/or program. In particular, the ‘XP Customer’ was the customer relationship manager, rather than any particular customer. This does not rule out ideas being generated by the programmers themselves and they certainly have a strong influence on how ideas are argued about and realised or not.

So, what can we say about where the requirements come from? Clearly, requirements do come from customers and from understandings of customers, and from understandings of the market, or ideas about the potential market, but they do not usually come directly from customers in the manner that might happen in many agile projects. There is no contractual obligation on W1REsys to deliver specific requirements for specific customers. However, depending on their precise relationship with W1REsys, a customer may have a specific influence on development and have their requirement incorporated into the product. This is most likely, if a new requirement seems to be of generic use, or it opens up new opportunities.

Requirements are produced because they seem to make sense in terms of a balance of ‘can be done’, ‘would be useful to our customers – and may have been requested by some’ and ‘would be good for the product (and therefore market) development'. During the programming phase these requirements become crystallized as the programmers determine exactly how they will be realised (or not) in code.

3.1.2. Unit testing. W1REsys carry out fully automated unit testing of their code. They do not do test driven development, and they do not test every method in their code. They employ a mixed method whereby sometimes the test is written before the code, other times a test from their bank of tests is adapted after coding to test the new code, or a new test is written. When a new piece of code is to be integrated with the pre-existing code base full regression testing is run. In all, the code base is reasonably well tested and bugs in the released system appear to be kept to a manageable level. There are some interesting inconsistencies in unit testing, problems with deciding on appropriate tests and problems with deciding on what an error is caused by, also some complicated and interesting categorization and counting practices.

3.1.3. Systems integration testing. The system integration and acceptance testing phase ends up being the time left between build and release. This phase is referred to by W1REsys simply as “testing”; we have introduced the term “systems integration testing” ourselves for clarity. Generally this phase is adhered to in terms of how much time is allotted although programming time eats into it, and it may well continue post-release (or at least bug fixing may well).

W1REsys use story and task cards for requirements but for testing they use lists on a whiteboard of ‘*we need to test this, this, this*’ (rather than ‘*we need to test this, in this way, and this will be are criteria for indicating the test has been passed*.’) As such the tests are produced during the activity itself The criteria for passing the tests are socially and situationally produced. The developers work together on the tests, with such things as “what seems sensible and possible given what time we have?” and “what do we know or think about users and use?” influencing the design of the tests and the setting of pass criteria. This might appear as complete ‘ad-hocery’ but given that they do not know how companies and the developers using the W1REsys product to design applications will use the enhancements they tend to define tests in sensible ways – looking to conduct ‘proof of concept tests’ for hypothetical situations of use, often using resources to see if what they have programmed basically works.

If and when end-users start using the development environment for specific purposes bugs may come to light and then be fixed and the end-user can (if required) be assisted specifically. This leads us to another interesting feature of integration and acceptance testing for W1REsys; a good part of their orientation during this period is to prepare demonstration materials for the release. The tests that have been passed can serve as demonstrations to assist customers and users in taking advantage of the new features of the system, they show how programming can proceed and also demonstrate some of the capabilities of the new features. Of course, this is also very important in recruiting new customers. For these reasons, much of the integration and acceptance testing is concerned with ‘proof of concept’ testing and documentation, rather than defect testing.

3.2. Examples of systems integration testing

We will discuss aspects of testing from two ‘iterations’ of development at W1REsys. In the first iteration a major part of the development focused on enabling end-users to write applications that could access web services. In the second iteration the major focus was the redevelopment of the "message push server".

3.2.1. Enabling the system to access web services: The development of a requirement. There had been some interest expressed by some customers in having this facility and some discussion amongst the team. The web services were not a 'requirement' that had been specifically specified by a customer. Rather, it had seemed like a sensible course of development for the future. It would be suggested to current and future customers that web services could well form a business requirement for them, and that W1REsys would have a demo to back this up. Consequently, Gordon (the customer relationship manager) had written "web services" as a 'story' for the team of programmers.

One of us sat in on the project planning meeting, just prior to development when the programmers scoped out and scheduled the tasks required to fulfil the story. Planning the story proved to be a complicated business, lasting all day. Planning was a cooperative enterprise involving all four programmers (Paul, Tom, Dale, Mark) with Paul (the experienced developer mentioned earlier) overseeing the organisation of the activity and doing most of the whiteboard work. It had been originally conceived that the web services could be accessed from the graphical user interface in the interactive development environment (IDE) of the system. The end-user could simply type in the URL of the web services they wanted to access, and press a button and a 'web services wizard' would access the services and download a list of them. The developer could then select the ones they wanted to use.

The 'theory' behind believing that this requirement was manageable was that W1REsys's IDE uses XML as the development language and that web services description language (WSDL), used for describing the available web services, is also XML based. During the course of the planning session it became clear that the task was not quite so straightforward as might have appeared. Both W1REsys and web services might use XML but their data elements, attributes and relationships were likely to be modelled in different ways, they would have different XML Schema Definitions (XSDs), which would mean that the translation of any given set of web services would have to deal with the differences in XSDs, making the task of producing the wizard more complicated.

Through discussion it became clear that the team did not currently have an answer to how complicated this issue was. One of the key issues was whether web services used a standard form of XSD. No one in the team knew for sure. In order to try and help the planning, various discussions ensued amongst them, books were consulted and web services accessed to look at their WSDL and XSD. During the course of the afternoon it was decided that they would go ahead with trying to develop the requirement but that the first task would be to assign two programmer person days to "investigate WSDL tools" to test the viability of the story. The end-result was that the 'requirement' was not fully defined but was 'good enough' for progress to be made.

3.2.2. Testing the web service functions. The requirement for developing access to web services had a 'specifically vague' quality by the end of planning. The question as to whether a wizard would work for all web services was left open, it was not supplied by a specific customer and it was scheduled to be investigated with the clear possibility that it might even be shelved completely. Clearly, this was not a situation in which it was possible to stipulate exactly what the test would be in advance and it certainly would not have made sense to spend some time planning out a series of tests that might never be needed.

At the end of the iteration the team had reached a compromise regarding the web services wizard. During development they had decided to develop two mechanisms for composing requests to hit web services. If it was a "primitive" service they could automatically generate the request using the wizard, and if it was a "complex" service (e.g. to collect a repeated series of records such as search results) the request could be programmed manually as XML.

Having no specific web services described in the requirements or requested by customers, the team decided that they would try to access both primitive and complex web services over a range of mobile devices. The main feature of the W1REsys product is that it can translate single programs to different forms of mobile device. As a result, the starting place for this 'testing' was a PPC and then secondly a (MIDP 2.0 conforming) mobile phone. The web services that they used for testing were free services provided by a service provider specializing in customer relationship management solutions, and a spell check service provided by an internet search company.

The companies provided full details of the (SOAP) requests required to access their services and also of the data that should be returned. Paul's job during testing was to write the code for the requests and then test whether he was getting the response to firstly the primitive requests, then secondly to the complex requests on both the PPC and mobile phone.

Paul began with the PPC and encountered a number of problems getting the requests to access the web services, which provoked amongst other things further work on their code. A central problem was that it was often unclear if tests failed because of a lack of response or errors in handling the response. Paul managed to sort out this problem by using a network analyser to view what was going on in the communication between the PPC and the web services provider. Eventually he managed to handle both simple and complex services.

When we returned the next day, Paul was working on exactly the same test but this time using a mobile phone. Again, he encountered problems, but this time they were harder to solve. The main problem here was that he could not use the same method to understand what was going on in the communication between the phone and the web services. As he ruefully remarked "mobile phones do not have telnet". He was using a combination of a real phone and several different phone emulators that he had installed on his workstation. However, he also remarked "emulators are only of limited use – if the application works on the emulator it probably works on the phone, if it does not, the result that it produces may make no sense at all."

Getting the web services to work was a drawn out process of trial and error with different members of the group involved in the troubleshooting efforts at different points. By the time Paul had sorted out the web services for the phone and the PPC it was late on Friday afternoon and the testing period was basically over. The documentation produced during the testing phase included a demonstration of using the web services used in testing.

3.2.3. Building a new message push server: The development of a requirement. During the testing phase of the iteration discussed in the previous section, the team had a conversation about the 'push server' in their application. The push server is responsible for 'pushing' messages to remote devices. In this section we summarize various conversations to do with development and testing of this server.

The first example illustrates the ways in which a problem gets formulated. This could also be seen as the construction of a 'requirement'. It should be clear from the example that this was not the first time the current push server had been questioned as there are references to previous events. Furthermore the iteration in which the new push server was constructed did not happen until 7 months after this conversation.

The conversation begins when Gordon returns from lunch and sits with them. He has been experiencing problems when demonstrating the existing push server to a customer. The conversations in this section are taken from field-notes rather than recordings and so are partially incomplete. We use '…' to denote speech we were unable to write down.

G *"Is it dying a death?"*
D *"It's okay for a couple of days …" He goes on to explain that it makes so many connections that it takes gigabytes of memory.*
P *"Is that server memory? You know they need to be doubling the size of the server… they should be distributing the load… you know, which is a bit of a cop out… but if they've got someone who understands Oracle then they should be able to role out to a couple of servers…"*
D *"… its difficult."*
D *"Well they were going up to 1000 users … the impression I got was they want to make it enterprise wide."*
G *"Enterprise wide sounds like a critical issue."*
D *Explains that this is from [Customer X] but "he doesn't seem that concerned."*
G *"Something doesn't sound right to me… you know last year I was getting phone calls at 6am… we've always known the push server is something we need to see to again and again… and you know we're going to get scarier customers than [Customer X]." He explains they might get a large repair company as a customer.*

The example demonstrates how issues or concerns become 'something we need seriously to think about dealing with' as they accumulate over time and are brought together conversationally by the team. The problem is known about, has a history, may give a negative impression to customers, and in terms of the future development of the system and in the light of potential new customers it is a prescient time to think about re-building the server.

The conversation developed in an interesting fashion. As we have stated, requirements are only fully known when they have been produced through programming. Therefore acceptance and integration testing is scoped, revised and so forth during planning, development and testing. As we can see here, this does not stop testing being considered early on:

G *"We've never figured out a way we can test it other than get someone with 500 users."*
D *"No, I can spike test but that's not a proper test…"*
P *"No, … you could actually write a program that makes several PCs make loads of connections."*

The discussion continues with the developers trying to come to some understanding of how numbers of users translate to numbers and timings for server connections.

D “Its not like you’d get 100 users at a time…”
G “But the reality of 700 users is … every three minutes it … and every 30 seconds if there’s something to push.”
P “Ok, so every 30 seconds is a push check and, no, every three minutes…”
G “That’s not a lot.”
D “That -is- a lot of connections.”
P “If you’ve got… then you’ve 20 a second doing a select on the server, if you assume an even distribution… you could have a peak of 600 but that’s unlikely.”
P “…It’s not complicated but… you’d have to have a maximum, … and you’d have to wait…”
G “Right, but 20 transactions a second isn’t a lot.”
P “No it’s not a lot, but … you’d have to …”
P “It means … buying a pool manager.”
G “Does Oracle come with a pool manager?”
P “Not one free…“
G “More than a couple of quid?”
P (laughs) “Writing a pool manager is not hard.”
G “So would that solve it?”
P “Alleviate it.”

The team were trying to scope the issue and set sensible parameters. They went on to discuss how there was a danger of losing customers to a rival company and that a potential new customer had up to 1400 users. Our excerpt finishes with them looking for inspiration for a new design:

G “How do [Company Y] do their code? Can we not nick it?!”
P “I don’t know who they are.”
G “[Name of Product Y]”
M “Aren’t they on their own gateway?”
G “… They have a push server effectively.”
P “Really?”
G “…so it has 100 million people connected to it. It’s alright you don’t have to go to 100 million straight away!” There is laughter.
G “But it would be good to know how much time …”
G “So are you confident if we had a connection manager that the problem would just go away? Or is it just one thing in the scalability?”
D “… I don’t know the way out of it.”
G “I’ll speak to [Customer X], and I’ll speak to [Customer Z] but the push server is likely to be three or four weeks work.”

This example illustrates a lot of the ‘preparatory’ investigative work that is part and parcel of code development in this company. The team are involved in figuring out whether they need to re-build the push server (how serious is the problem? who would use it? what are the potential pitfalls of doing nothing? etc.), how they would need to do this, what sort of resources they could draw upon, how long the project would take and how it would be tested.

3.2.4. Testing the message push server. The push server iteration was an unusual one for W1REsys in that it was 3 months in length, much longer than the above estimate, and much longer than their usual development cycle of 6-8 weeks. There was a general aversion to longer iterations – for amongst other reasons being ‘non-agile’ and being harder to manage – but it was clear to the team that rebuilding the server would take a longer time.

For the testing phase they had built a ‘test harness’: the message server was installed on Paul’s machine and they had devised a way to simulate messages being sent from each of their workstations to the server and back such that they could test whether the messages were being sent and received successfully. Paul was coordinating the test sessions which required the others (Mark, Tom, Dale and Gordon) to configure their machines for the test and initially to get them to send out 1000 messages each, one after the other. As we joined them they were just making sure that everyone was correctly set up for the test. To monitor progress Paul has a ‘push server monitor’ up on screen. This allowed him to view the progress of the ‘messages’ as they came from each machine. He provided a commentary of the messages coming in. After minor adjustments they came through successfully:

P “5000 – Amazing! Now I’m going to send one message to all 5000 back.” All messages are dispatched: “5000 calls on the API!”

As the messages get sent back the programmers comment on them coming through. Again this is successful so, after a joke about testing being finished, they decide to double the amount of messages:

M “Mine are coming through. Got 700, sequentially.”
T Confirms his are also coming through.
P Looks over his system, then says to G “1000.”
G “So that’s scalability testing done? … That’s 5000 messages in a minute, will we try with more messages?”
P “I’m thinking about trying 10000, so we have to change to 2000 messages each … I think my machine will potentially shit it with 10000 sockets, we need to change our offsets. Double them both, all of you.”

In the second test only 9985 successful connections to the push server are managed, meaning 15 had been denied. This raised concern, but was offset by Dale noting:

D “But this is just the test harness.” (i.e. it will work slightly differently in a real situation).

They then dispatched 2000 messages back to each client. It took 30 seconds to dispatch 10000 messages.

As the test ran they talked about further tests particularly one on message prioritisation. This talk was interrupted as problems occurred:

P *"We've still got 15 users missing." He spots in the 'Push Server Monitor' that D is unregistered.*
T *"I've only got 200 and something through."*
P *"Waiting queue monitor pointer exception, exception in thread. There's a problem with the push server, it broke!"*
M *"It's got to be something straightforward but it could be hard to find." He checks through files but there is "nothing obvious right now … Why would it do that? It doesn't make sense."*
G *"My connections died – could it be anything to do with that, maybe?"*
P *"One message failed and caused the whole thing to stop."*
M *"We want to sort the whole thing out higher up, it should still continue if the message fails rather than bothering to try and understand why the message failed. In reality it should just try and send it again."*
P *Looks at the code "There's nothing on this thread to handle general exceptions."*

The test failed, and the team realised that this was because failed messages were causing the whole queue to fail, which lead them to consider why this was the case. After some examination of code Mark suggested that they should focus on a means of keeping the queue operating in the face of failures by putting failed messages to the end of the waiting queue. They proceeded to devise a method for doing this, tagging failed messages with a lower priority that placed them to the end of the queue so that other messages behind it would still be delivered.

After 'solving' this issue the testing naturally moved onto message prioritisation. A third of messages were tagged as priority '1', a third with '2' and a third with '3'. The delivery would work according to the principle that three '1s' will be delivered followed by two '2s', then one '3'. As the messages are delivered, when all the 1s are delivered there will still be 2s and 3s, and eventually the queue should end with the delivery of the remaining 3s. When they set the test in motion they did not get the messages coming through in the right priority. Dale explained:

D *"We need to use a round robin to take 3 priority 1s, then 2 priority 2s, then 1 priority 3 off the queue because otherwise priority 1 might always be top of the queue, as a failed priority 1 would always go to the top of the queue. But anyway, we would expect to see more priority 1s coming through, but this is not the case."*

This lead to a number of investigations as to why prioritisation was failing before the team focused in on the code for the round robin (rr) queue:

P *Looks at the code "This round robin queue doesn't look right."*
D *"No, it isn't."*

Although Paul and Dale felt they had basically located the problem they could not find out exactly what was causing it and it was only after Dale had spent some time walking through the code that he managed to find that it was the 'peek and remove' procedure that was operating on the 'rr counter'. It was reading (peeking – a check of whether the number matched) and not removing the item it was peeking at, i.e. not taking the right thing off the queue. Dale explained how he had verified his solution:

D *"I found the peek problem through doing a code walkthrough, the pattern of the messages currently being delivered served as the verification… it's often easier just to walk through the code when a problem arises. The rr counter was being modified by the peek and remove. It was saying it was looking in one place when it was looking elsewhere."*

Paul and Dale then ran the test again, and it was successful this time, so they did a full build. This finished the first of three weeks of testing for the push server. They now knew it should be able to handle at least 10000 connections at once, and also be able to handle prioritisation. This meant that they could specify this capacity and performance to current and future customers and also demonstrate it.

4. Lessons learned

In his seminal work on the 'art' of software testing Myers makes a strong case not only that software testing is often misunderstood but also that the determinants of successful software testing have little to do with purely technical considerations but are best seen as issues of economics and psychology. Whilst we agree with Myers' general argument that software testing should not be regarded as merely a technical issue, we now want to draw on our empirical findings in order to reframe his formulation of the software testing problem, suggesting instead that testing problems are primarily organisational in character and consequently that they may be best addressed, if not resolved, by organisational and inter-organisational means.

So, for example, what Myers identifies as economic issues, of prioritising resources and costs, are easily reframed as a classic example of Garfinkel's

[11] ‘administrator’s problem’ since decisions on whether the time and effort are justified are essentially and contingently organisational rather than purely economic. We also suggest that other software testing problems, seen as predominantly ‘psychological’ by Myers appear, from our empirical evidence, to be better couched in terms of a range of mundane organisational issues connected to planning, tracking, coordinating etc. Just as Garfinkel in his phrase ‘good organisational reasons for bad organisational records’, documents the range of good organisational reasons behind recording systems that are generally less than perfect, reasons that work to the advantage of the organisation concerned, so we also suggest a range of organisational concerns that complicate the testing process, a process that is characteristic of many ‘wicked problems’ [21]; and for which there are, perhaps, only ‘satisficing’ solutions. The use of scarequotes around ‘bad’ software testing in our title is not a judgement on the methods or the company that was the focus of our research but an acknowledgement of the everyday reality of real world, real time testing.

One overwhelming feature of everyday, mundane organisational reality for all companies is how to deploy limited testing resources to find faults or design problems, or to see if their system operates in the desired manner and meets customer needs. The testing problem is exacerbated by organisational and commercial pressures for rapid delivery of software, increasing complexity, more demanding quality requirements and volatile system requirements.

For small product development companies, the problems of resource deployment are even more acute. W1REsys are a start-up company working on venture capital, developing a ‘generic’ system. As such, they are designing for a range of users and potential future users. Their priority is survival which means firstly, generating cash flow from customers and, secondly, building and maintaining customer relationships. We believe that these considerations are likely to be shared with most small software companies so our conclusions here are not specific to W1REsys.

W1REsys are not ignorant of best practice in software testing (test case design, test coverage, regression testing, etc.) but, for good organisational reasons, chose not to adopt this but to orient their testing to meet organizational needs. Their driving priorities are:

1. *The dynamics of real customer relationships*. Gordon, the customer relationship manager, meets customers regularly and feeds back information from these meetings to developers. The feedback is not ‘hard’ requirements but more general thoughts on how customers anticipate using the product.

2. *Using limited effort in the most effective way*. As discussed, testing effort is very limited and, sometimes, effort has to be diverted to more pressing concerns such as emergency bug fixes for major customers.

3. *The timing of software releases*. Software releases are timed around the needs of major customers and the features included in a release are influenced by these customers and the development effort available.

4. *The need to ‘grow the market’ for the system*. As well as satisfying existing customers, it is important to anticipate what customers (both existing and new) may wish in future. The previous discussion on linking with web services is a good example of this.

These priorities influence different aspects of the testing process namely: requirements testing, test coverage, test automation and test scenario design.

4.1. Requirements testing

In principle, requirements tests should be complete insofar as all aspects of all requirements are tested for both normal and exceptional operation. In practice in W1REsys, the fluid, negotiated nature of requirements makes this impossible. Requirements and therefore system integration tests are derived as a judgement from a variety of sources. Tests are related to requirements – as we have seen, especially in the push server example, tests are scoped alongside, and as a part of deriving a requirement – however, even when a requirement has been realized the question of a sufficient test is still open. An ‘adequate’ test of any requirement in W1REsys’s case means taking a consideration of how and when it is likely to be used and which customer will use the associated feature.

When a requirement is proposed, only the fullness of time will tell whether that requirement is a sensible or successful one. For example will a requirement that happened to take up a long time to develop only serve a single user or will it open up a new market? The same is true for the adequacy of testing; future uses and customers may show testing to have been inadequate.

From our examples it was clear that the higher capacity push server was something that customers wanted now, however, when we discussed the web services facility with Paul after the iteration he told us that it might be 6 months or so before any customer used them. This initially surprised us. However, we realised that a lot of W1REsys’s requirements are for the future, and as such involve predictions about what will be desired in the future. They had to anticipate customer needs and what might sell the system to new customers. The role of testing in this case is to demonstrate the requirement rather than to discover defects in its design or implementation.

4.2. Test coverage

For W1REsys it is clearly impossible to define what complete testing might be, they simply do not know at the point of testing just what all the possible situations of usage and uses their system will be put to. Different customers (and potential customers) will want to build different applications for different mobile devices, and will have different requirements for managing communications with those devices. In this way testing can never have complete coverage.

From the examples discussed it is clear that testing could have been more extensive. However, the need to retain existing customers and the target release dates for the software restricted coverage. More attention was paid to testing features that large customers were likely to use in the next release. The key issue was not how extensively a feature had been tested but whether or not it been tested sufficiently for the customer to make effective use of it. W1REsys's approach is thoroughly pragmatic – why waste resource on something that cannot be well defined in advance?

4.3. Automating testing

Using fully automated acceptance tests entails a particular style of development that produces 'testable' code. It also entails a customer relationship whereby the customer is thinking about their software is 'testable' ways. Could automated testing 'solve' or pre-empt the organizational issues we recognize? Automated tests are unambiguous – there are clear criteria established for a successful test and the test results are checked automatically against them. The 'XP Customer' at W1REsys was the customer relationship manager, and so responsibility for requirements and testing lies within the company. The notion of a test as a 'contract' between the customer and the software developers makes little sense in this situation. They do not wish to define their requirements to the level of detail where it would be possible to construct an unambiguous, automated test. Furthermore, the need to meet release dates meant that the customer relationship manager re-interpreted what is meant by 'passing a test'. If tests had 'failed', there would still have been the need for discussion and interpretation of the test results.

Our results confirm many of the suggestions made by Berner et al [3] to be sensible, and confirm that it is unrealistic to expect automated tests to fully replace manual tests.

4.4. Test scenario design

The development team at W1REsys informally and cooperatively derive scenarios in an on-going fashion throughout development and testing. They are seen to be involved in imagining how different users (both the customer as part of their business, and the end-user as a programmer) would use the system. For example, they envisage what sort of web services, accessed via what kind of device a customer might use, or, how many connections would different users require to the push server, over what time and with what prioritisation? Developers' informal production and talk about scenarios helps the design crystallize.

The big questions for W1REsys lies in the adequacy of the scenarios – is their research good enough, do they really understand their customers? The nature of their operation which includes a number of different customers makes it difficult for them to construct generic scenarios that include the requirements of different customers. Their test scenarios therefore tend to be fragmented and incomplete. Chillarege [6] points out the methodology of developing user scenarios and using enough of them to get adequate coverage at a functional level continues to be a difficult task and our findings further compound such problems.

5. Conclusion

Our studies have shown that, for this type of small company involved in software product development, there is a disconnect between software testing research and practice. Software testing research has largely focused on making testing 'better' in technical terms – improving the design of tests, designing tools to support testing, measuring test coverage, etc. In practice, these don't help as the key issue is how to design tests that are most effective in satisfying organizational needs and that minimize the effort and time required to demonstrate that software is 'good enough'.

Our studies of W1REsys have convinced us that the agenda for software testing research has to be extended to address the relationship between the organization and the testing processes. Alongside the existing hypothesis driven case studies of methods and technologies, Software Engineering's programme of empirical research on testing can benefit from studies of work that seek to understand testing as it happens.

Our study has been of a single company, and whilst there are general lessons to be learned here, further studies of work in similar or dissimilar companies will help in developing an understanding of the organisational rationale for how software testing is practiced.

6. References

[1] Bach, J., "Risk Based Testing. How to Conduct Heuristic Risk Analysis", *Software Testing and Quality Engineering* November/December, 1999. pp.23-28.

[2] Bashir, I., and A.L. Goel, *Testing Object Oriented Software*. *Life Cycle Solutions*. Springer, New York, 1999.

[3] Berner, S., R. Weber, and R.K. Keller, "Observations and Lessons Learned from Automated Testing" Proc. International Conference on Software Engineering ICSE'05, St. Louis, 2005. pp.571-579.

[4] Blythin, S,, J., Hughes, S., Kristoffersen, T. Rodden, and M. Rouncefield, "Recognising 'success' and 'failure': Evaluating Groupware in a Commercial Context", Proc. Group'07, ACM Press New York, 1997. pp.39-46.

[5] Button, G., and W. Sharrock, "Occasioned Practices in the Work of Software Engineers", IN Jirotka, M., and A. Goguen, (Eds.) *Requirements Engineering. Social and Technical Issues*. Academic Press, London, 1994. pp.217-240.

[6] Chillarege, R., "Software Testing Best Practices", *IBM Research Report RC 21457, Log 96856*. IBM Research, York Town Heights, 1999.

[7] Collins, H. "Public Experiments and Displays of Virtuosity: The Core Set Revisited", *Social Studies of Science*, 18 (4) 1988. pp.725-748.

[8] Crispin L., and T. House, *Testing Extreme Programming*. Addison-Wesley, New Jersey, 2003.

[9] Eisenstadt, M., "My hairiest bug war stories", *Comm. ACM*, 40 (4), 1997. pp.30-37.

[10] Evans, M.W., *Productive Software Test Management.* John Wiley & Sons, New York, 1984.

[11] Garfinkel, H., *Studies in Ethnomethodology*. Prentice Hall, New Jersey, 1967.

[12] Harrold, M.J., "Testing: A Roadmap", Proc. International Conference on Software Engineering ICSE'00, Limerick, 2000. pp.61-72.

[13] kaner, C., "Software Testing as a Social Science Problem", Canadian Undergraduate Software Engineering Conference, Montreal, Canada, 2006.

[14] Kaner, C., J. Bach, and B. Pettichord, *Lessons Learned in Software Testing*, John Wiley & Sons, New York, 2002.

[15] Kim, J-M., A. Porter, and G. Rothermel, "An Empirical Study of Regression Test Application Frequency", Proc. International Conference on Software Engineering ICSE'00, Limerick, 2000. pp.126-135.

[16] Knuth, D.E. "The Errors of TeX", *Software-Practice & Experience*, 19 (7) 1989. pp.607-685.

[17] Martin, D., M. Hartswood, R. Slack, and A. Voss, "Achieving Dependability in the Configuration, Integration and Testing of Healthcare Technologies", *Journal of Computer Supported Cooperative Work*, In Press.

[18] Mosley, D.J., and B.A. Posey, *Just Enough Software Test Automation*. Prentice Hall, New Jersey, 2002.

[19] Myers, G.J., *The Art of Software Testing*. John Wiley & Sons, New York, 1976.

[20] Ramler, R., amd K. Wolfmaier, "Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost" Proc. Automation of Software Test AST'06, Shanghai, 2006. pp.85-91.

[21] Rittel, H. and M. Webber, "Dilemmas in a General Theory of Planning," *Policy Sciences* 4, 1973. pp. 155-159.

[22] Runeson, P., *A Survey of Unit Testing Practices*. *IEEE Software*, July/August 2006, pp22-29.

[23] Stringfellow, C.V., and D.L. York, "An Example of Practical Component Testing", *Journal of Computing Sciences in Colleges*, 19 (4) 2004, pp.203-210.

[24] Talby, D., A. Keren, O. Hazzan, and Y. Dubinsky, "Agile Software Testing in a Large Scale Software Testing Project", *IEEE Software*, July/August 2006. pp.30-37.

[25] Whittaker, J.A., "What is Software Testing? And Why is it so Hard?" *IEEE Software*, January/February 2000. pp70-79.

[26] Whittaker, J.A., *How to Break Software*. Addison Wesley, Boston, 2003.