Poor defects BUG me!
Peter Morgan – Nicemove Ltd
Belgium Testing Days 2010, Brussels

Biography: Peter Morgan is a testing professional who has been involved in the ICT industry for more than 30 years, and worked in the freelance marketplace for much of that time. Peter was one of the first tranche to sit and pass the (old-style) ISEB Practitioner's Certificate in Software Testing, and he continues to play an active part in testing examination schemes in the UK and Europe (for example as a question writer). He is one of the co-authors of the ISEB book based on the ISEB/ISTQB Foundation Certificate in Software Testing syllabus. Peter describes himself as "a grass roots, coal face, hands-on, defect raising, problem solving, test case generating, data creating, regression running, document reviewing tester". As an enthusiastic speaker and author, Peter tries to base his output on practical experience, attempting to relate fine sounding ideas back to how it will affect Joe or Jane Tester in their everyday working lives in the war of attrition that we call software testing.

Contact details: morganp@supanet.com **Website:** www.nicemove.biz

Abstract: The success of a project probably depends in part upon the effectiveness of its defect tracking, and possibly on the quality of defects raised. In this paper Peter uses a three-way classification of defects that he has found useful: good defects, bad defects and poor defects. A good defect is a genuine problem and one that is resolved quickly (because it is clearly formulated), whereas a bad defect is hard to isolate (perhaps seldom able to be reproduced) and/or hard to fix. On the other hand, a poor defect will not get fixed rapidly as it is vague, does not provide all of the required information and/or is unhelpful not only to the developers but also to the project in general.

There are many types of defect tracking 'tools' available; some bundled with Test Management tools, others propriety stand-alone application, free-ware or open source. One of Peter's favourites tracking systems is Excel; simple to use, requires no training, and easily able to produce meaningful statistics. It is of particular benefit where there are small project teams. However, the important point is not which defect tracking tool or system is used, but that a strong and robust process is in place. The IEEE 1044 standard can be a good place to start, with the concept of a defect life-cycle. A defect is raised, someone does something to the application or testing artefacts, marking the defect as 'done' or requiring no action and in due course, it is closed.

The final and largest part of the paper concentrates upon improving the defects that we write, or in how testers view the overall defect process and Peter has numerous tips, suggestions and recommendations. A minority of these have been included in some previous material [Morgan 2002], [Morgan 2007], but the whole benefits from being described together.

Table of Contents

Introduction

Goals

The prime aim of this paper is to encourage you to think defects, and raise the standard of the defects that you raise or see raised on your specific projects, both now and into the future. There can be no magic list that will help everyone in every situation, but my hope is that several of the hints, tips and tricks will be useful in any working environment It is just that the 'several' will change depending upon, for example, the vertical market segment you are operating within, the

software development methodology, the cultural background of the company or the experience of those undertaking the testing.

I am passionate about testing and passionate about defects. If the detail that follows causes you to rethink your ways of acting and in particular the raising of defects and the defect process, then it has achieved its primary purpose. That is true, even if you seriously consider your former practices, and after due consideration, return to these with no changes.

The long introduction is necessary in order to place the "Defect hints, tips and tricks" section into context, and to enable anyone (regardless of testing experience or indeed whether they are a tester at all) to benefit from this important section. Those contemplating moving straight to the final section would be advised to think again.

Overview

It is not expected that readers of this paper will necessarily be experienced testers; it is aimed at anyone. The first sections are an introduction, discussing, for example, how we name problems that we find in software, what is testing and some examples of interesting or amusing defects. Thereafter follows a short synopsis of the defect life cycle, before the idea of **good**, **bad** and **poor** defects is introduced, something the author has found useful. It is only when this groundwork has been laid that substantial benefit can accrue to those with comparatively little testing experience or background. It is the "Defect hints, tips and tricks" where most benefit will be found. The aim of this paper is that anyone can read it and gain insight, understanding, and, yes, useful and practical suggestions.

What do you call 'them'?

On the projects I have worked on, in a variety of companies and spanning diverse vertical market segments, the name given to "issues" has varied tremendously. Here are some that I have come across, but I am sure that you can add to this list:

> Issue, Anomaly, CFR, Incident, Bug, Defect, Tracker, Trouble Ticket, Problem, SPR, Observation, Fault Report, PinICL

What you do call them in a particular workplace does not really matter, as long as everyone knows what you mean by the term. The real problem comes when the same word is used with different meanings, or different words have the same meaning to different people [and they do not understand the other's terms]. Wherever I work, I adopt the term for 'issue' that is in use there, whilst making it clear what I mean by the chosen term. Three used in the company where I am currently on contract (and used as-if they are the same words) are 'defect', 'bug' and 'observation'.

We need to have a common term in our work-places, and avoid the Humpty Dumpty syndrome: 'When I use a word,' Humpty Dumpty said, in rather a scornful tone, 'it means just what I choose it to mean — neither more nor less.' [Carroll 1871]

A simple question

Do you believe in defect-free software? In one sense, I feel that we cannot be involved in testing if we do NOT believe in the concept. Much of our activity, ultimately, is aimed to reducing the number and severity of defects found, in essence 'making the software better and better'. Of

course, our intervention as testers is not direct, but provides information to others to enable them to rectify problems. My position is that I passionately believe in defect-free software. However, the not inconsiderable drawback is that it is impossible to know whether you have achieved this.

Nothing new in testing

Many would say that testing began in 1979 with the publication of what is still considered as a seminal work [Myers 1979]. Since that time, pre-existing and new techniques have been defined and refined, processes written and standards proposed. However, the essence of software testing is still unchanged: to critically examine software-related works products, report possible problems, and see these resolved (which may or may not require changes to items, including the testware).

Definition of testing

It is not my intention to provide yet another definition of testing, so don't expect a polished summary. This is merely a form of words that I have found helpful, and use of these words has almost single-handedly played a significant part in securing contract positions for me on at least three occasions.

> Testing is the systematic and methodical examination of a software product using a variety of techniques, with the express intention of attempting to show that it does not fulfil its desired or intended purpose. This is undertaken in an environment that represents most nearly that which will be used in live operation

The testers paradox

Testing has twin aims which are incompatible, and give rise to "the tester's paradox". As testers we seek to find problems, yet are part of the production of working software. If we succeed in one of these aims, we fail in the other. THAT is the paradox.

Best / worst defects

If a group of doctors get together, they don't spend time talking about patients that they have killed. Rather it is interesting cases that perhaps initially perplexed them, but where eventually, through research, dedication and sometimes just plain luck, a correct diagnosis was given, the patient responded and is now walking around alive!

Testers are different. Quite often it is the one that got away that enjoys the limelight, and rightly so, as these little gems can teach us, and help to catch similar defects into the future. I have several to add to that mix

On-line book seller software upgrade

A well known on-line book seller (as this is Belgium, let us call them "Meuse.com", for example) had a major software upgrade, which was duly tested before deployment. As with some other very complex systems (perhaps nuclear power stations, or the opening of a new airport terminal – Heathrow Terminal 5), it was not possible to test everything together before entering production

– it is not practical to build a test system with a test book warehouse, a test web-site undertaking $100,000's of business every hour, and huge volumes of customers 'just browsing', etc. Parts were tested in isolation and then the cut-over to the new software took place. All went well until it was noticed that the books that had been ordered and packaged for despatch had blank labels on them. There was no indication of where the goods were to be despatched, or to whom. Now that was a slight problem!

Australian cookbook

Recently a cookbook was produced in Australia, and the contents were proof-read during the editing process. Proof read by people, with an eye for detail. After this had been completed, the text and pictures were auto-formatted (for page layout, position of the pictures in relation to the text, building of the index, and table of contents, etc). Part of the final checks involved automated spellchecking. Unfortunately what was not picked up was the fact that this resulted in two 6-letter words being inter-changed; pepper and people. After the mistake was spotted, 1,000's of books had to be pulped, for the as-printed text now showed: "Sprinkle with freshly ground black people"

WORD 95 spell-check problem

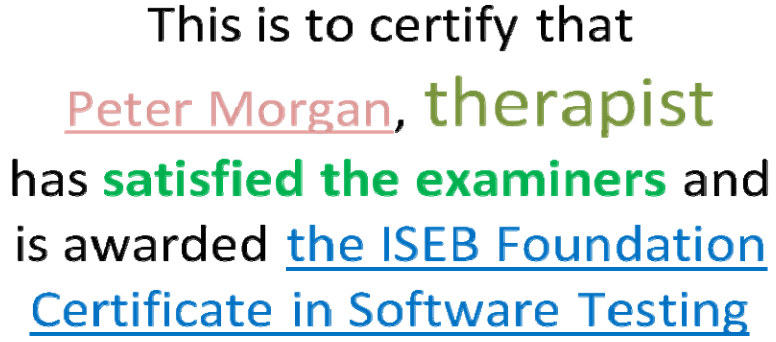
A spell checker highlights words where the spelling is not recognised, and makes a suggestion for possible alternatives. In WORD 95, if you spell-checked a document with the words "I wish Bill Gates were dead" in it, this would be highlighted as a spelling mistake. The suggested correction displayed was "I'll drink to that".

Certificate printing

In the late 1980's, there was a certificate printing utility that was available. Using a simple user interface, the operator entered 4 pieces of information, and a certificate was printed. The required information together with example values is as follows;

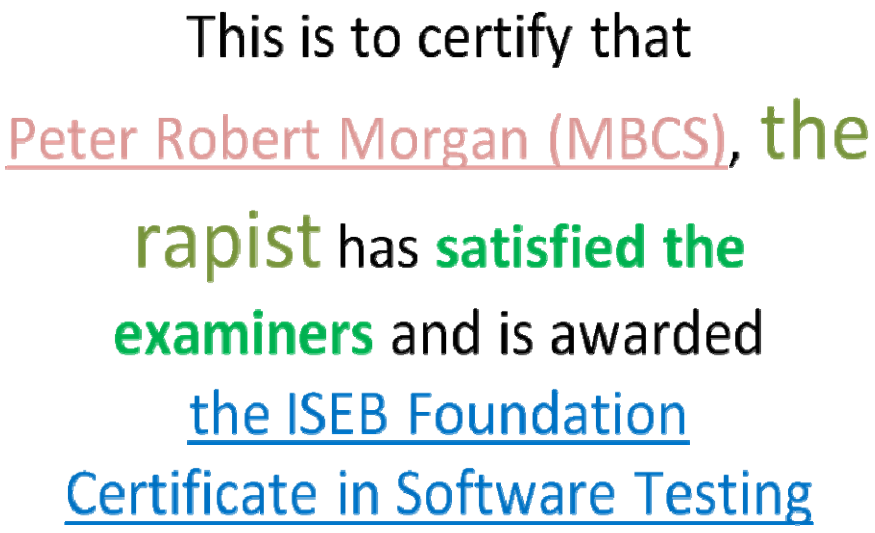
Description	Values I used
1) Name	Peter Morgan
2) Occupation	therapist
3) What had been done	satisfied the examiners
4) The 'prize' that was awarded	the ISEB Foundation Certificate in Software Testing

This produced the fine-looking certificate shown following, with the user-defined information in different colours for illustrative purposes only (and font sizes / font emphasis, to assist any that find colours difficult to distinguish).



This certificate meets the user requirement; you get the idea. However, if instead of entering "Peter Morgan", I wish to use my full name, and detail that I am a member of the British Computer Society, the use of "Peter Robert Morgan (MBCS)" produces a very unfortunate splitting of words over a line break.

The occupation has moved from 'therapist' to 'the rapist', clearly not what was intended.



Excel 2007

There is a problem with Excel 2007, when using 'Paste Values'. This can be shown with a simple spreadsheet, where sales targets are entered, with an indication of whether that target had been met.

	A	B	C	D	E	F	G
1	Week	Target	Met	Date		Paid A	Paid B
2	1	1200	Yes	16/12/2010			
3	2	600	No	29/12/2010			
4	3	500	Yes	29/12/2010			
5	4	800	Yes	07/01/2011			
6	5	900	Yes	14/01/2011			
7	6	400	Yes	28/01/2011			
8	7	1500	No	28/01/2011			
9	8	800	No	04/02/2011			
10	9	1220	Yes	11/02/2011			
11	10	650	No				

Column C now has a filter applied, with only those weeks where the sales target has been met shown, detailed below.

	A	B	C	D	E	F	G
1	Week	Target	Met	Date		Paid A	Paid B
2	1	1200	Yes	16/12/2010			
4	3	500	Yes	29/12/2010			
5	4	800	Yes	07/01/2011			
6	5	900	Yes	14/01/2011			
7	6	400	Yes	28/01/2011			
10	9	1220	Yes	11/02/2011			

The date when the bonus has been paid is entered into cell F2, and using copy and paste, applied to the appropriate entries in column F. However, this removes the highlighting on this column that had been used for some of the rows.

In order to get around this difficulty, ‘paste values’ was used. The date the bonus was paid was entered into G2, and copied. Cells G2 to G10 are highlighted, and paste values used. Now the formatting of the colour-highlighted cells is preserved.

	A	B	C	D	E	F	G
1	Week	Target	Met	Date		Paid A	Paid B
2	1	1200	Yes	16/12/2010		15/02/2011	15/02/2011
4	3	500	Yes	29/12/2010		15/02/2011	15/02/2011
5	4	800	Yes	07/01/2011		15/02/2011	15/02/2011
6	5	900	Yes	14/01/2011		15/02/2011	15/02/2011
7	6	400	Yes	28/01/2011		15/02/2011	15/02/2011
10	9	1220	Yes	11/02/2011		15/02/2011	15/02/2011

The only difficulty now is that some of the cells on rows where the sales targets have not been met are shown as having the bonus paid. The previously applied filter is removed. Compare column F (where ‘paste’ was used) and column G (which resulted from ‘paste values’)

	A	B	C	D	E	F	G
1	Week	Target	Met	Date		Paid A	Paid B
2	1	1200	Yes	16/12/2010		15/02/2011	15/02/2011
3	2	600	No	29/12/2010			15/02/2011
4	3	500	Yes	29/12/2010		15/02/2011	15/02/2011
5	4	800	Yes	07/01/2011		15/02/2011	15/02/2011
6	5	900	Yes	14/01/2011		15/02/2011	15/02/2011
7	6	400	Yes	28/01/2011		15/02/2011	15/02/2011
8	7	1500	No	28/01/2011			15/02/2011
9	8	800	No	04/02/2011			15/02/2011
10	9	1220	Yes	11/02/2011		15/02/2011	15/02/2011
11	10	650	No				

Defect Life Cycle

Defect life-cycle is clearly expressed in IEEE 1044-1
A defect is

- Raised
- Something happens to it
- (Eventually) it is closed

The four stages in the ‘classic’ defect life-cycle can be remembered by RIAD – Recognition Investigation Action Disposition. If you want to remember this, think of the capital of Saudi Arabia – Riyadh.

Three defect types

In discussing defects, I have found a three-way classification useful; **Good defects**, **Bad defects** and **Poor defects**

These categories are not necessarily what they seem, so I had better explain

- Good defects – clearly define the problem and enables prompt diagnosis and resolution
- Bad defects – may clearly define the problem, but a difficult matter to fix (perhaps because it is difficult to isolate)
- Poor defects – ill defined problem and/or insufficient information

It is possible to be both a Bad defect and a Poor defect. The difficulty here is that the "badness" can be masked by the "poorness", meaning that the real reasons why the application behaved in an unexplained way cannot be investigated.

We have all probably encountered Bad defects. I maintain that every defect has a cause: there is no such thing as an un-reproducible defect. All defects are repeatable at will – providing that we have sufficient information. The 'sufficient information' is often the problem – we don't know all the facts, and when items are relevant and what are not. If we have the same starting conditions, with all relevant factors and influences as it was when the problem first occurred, then the defect could be reproduced.

I want to give an example of a 'bad' defect, one that was very difficult to isolate and hence provide a remedy. Once the cause was identified, in this particular case, the resolution followed swiftly. The defect showed itself in the Post Office automation project on which I worked. There were 19,000 Post Offices in the UK (at that time), and each was both a free-standing Windows-based office, and linked to the central database. The latter had transaction-by-transaction details of what was sold or paid out at each counter in each Post Office.

Communication was via ISDN line, and each Post Office would dial the central system every 30 minutes to see if there was any data to come down to that particular office (future dated prices changes, or new products that could be transacted), as well as sending any recent transactions back to the centre. In effect, the individual offices would ask the central system "have you got anything for me", before providing any information that had not been hitherto sent to the central message store. An overriding constraint was "have I sent or received anything in the last minute", and if the answer was no, the line to the central database was dropped.

Each Post Office had their computer system operational 24 hours each day, as it was overnight that data changes happened. Each day the application would close and restart at 03:00, but the Windows-based terminals would be permanently switched on. The difficulty was that some Post Offices got into a state of permanently communicating with the central system. The line would not drop, and data was not being transferred. The quick and dirty solution was to reboot the local machine, and at this point, everything returned to normal.

Not every Post Office was affected by this problem, and there seemed to be no rationale behind the instances. If there had been a hardware problem, meaning that the machine had to be swapped out, the Post Office did not experience this problem for a number of days after the hardware swap (if at all).

Eventually, the cause was found. The office terminals used a comparison of what the 'time' was and what the 'time' was now, when waiting to see if data had been transferred in the last minute.

The start time and current time were both stored using a 32 bit register to store the time (down to $^1/_{100}$ of seconds). Approximately every 47½ days, the 32-bit register would "roll over the top", from all 1's to all 0's. The time comparator was comparing a very high number with a very low one. Many offices would have had a system reboot before the 47½ days was reached, perhaps because of a printer problem, or because the Post Master felt like it! If there had been no system reboot in over a month and a half, the problem could occur. It was either this problem or the seemingly unrelated problem where an office stopped dialling the central computer every 30 minutes. This was indeed a very hard problem to solve, and a fine example of a 'bad' defect.

Writing better defects

General

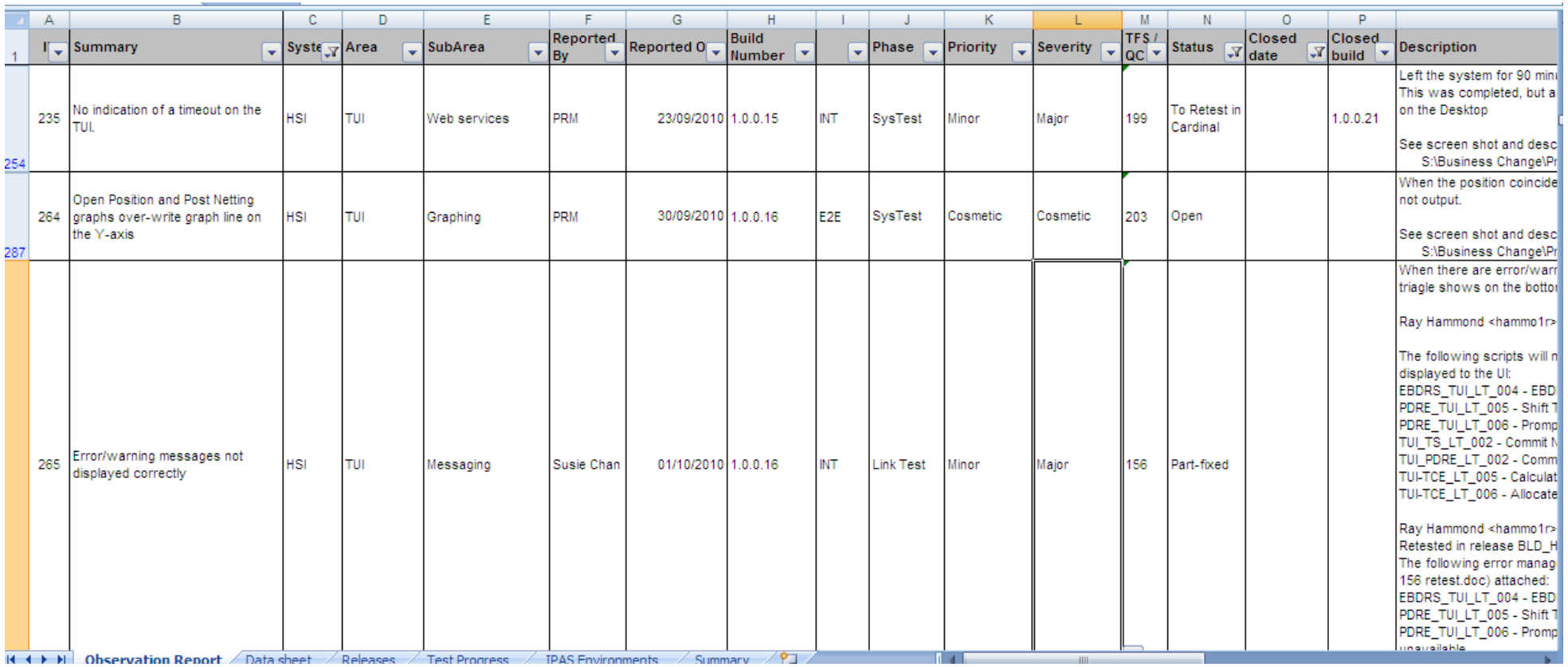
Have you ever worked on a project that has a poor or even non-existent defect tracking system? My first project had a paper-based tracking system, with five different coloured pages of the carbon impregnated multi-part form. [If you are interested, it was white, blue, red, green and yellow, although memory lets me down on the order of these colours]. It was not always clear which part was sent to different parts of the organisation, and misrouting was frequent. Defects can and did get lost, and the originator was not kept informed of the progress on his problem. The industry has moved on a long way since the early 1980's, but we need to make sure that defect tracking is not stuck in the past. Multi-site working is increasingly common, even across time-zones. A large project will stand or fall on its defect tracking mechanism. A poor defect tracking system does nobody any favours; at best it will hinder the project and at worst, cause it to fail.

The driving force behind this paper is that the standard of the defects that we raise can also have a significant effect on the projects we work on. If we can raise the quality of our defects, we can contribute significantly to improved quality of the software. Better defects will be fixed faster and more efficiently. It also means that defects get fixed, rather than sidelined because they are badly worded, contain insufficient information or are rude and offensive.

You may be surprised how badly some companies handle defects, with no central tracking, but with each project having its own. If there is not a central tracking system on a project, I introduce my own, often using Excel. Why Excel? It is cheap (i.e. free), easy to use, requires very little training, is flexible and a simple defect tracking system can be implemented in 10 minutes. It is also quite easy to produce some useful and thought-provoking statics.

The information to be inserted for each defect will depend in part upon any particular defect tracking tool that is used. Some proprietary defect tracking tools (e.g. Quality Center) allow input fields to be user-defined. I am not going to list information that should appear on a defect report – there is plenty of material already available on this. An example page from one of my Excel defect spreadsheets follows. If the column heading are helpful to you, use them, but your business needs could be different to mine, and the list may contain both insufficient columns and ones not required for your company's particular needs.

Example defect tracking using Excel



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	
1	I	Summary	Syste	Area	SubArea	Reported By	Reported O	Build Number		Phase	Priority	Severity	TFS / QC	Status	Closed date	Closed build	Description
254	235	No indication of a timeout on the TUI.	HSI	TUI	Web services	PRM	23/09/2010	1.0.0.15	INT	SysTest	Minor	Major	199	To Retest in Cardinal		1.0.0.21	Left the system for 90 min(This was completed, but a on the Desktop See screen shot and desc S:\Business Change\Pr
287	264	Open Position and Post Netting graphs over-write graph line on the Y-axis	HSI	TUI	Graphing	PRM	30/09/2010	1.0.0.16	E2E	SysTest	Cosmetic	Cosmetic	203	Open			When the position coincide not output. See screen shot and desc S:\Business Change\Pr
	265	Error/warning messages not displayed correctly	HSI	TUI	Messaging	Susie Chan	01/10/2010	1.0.0.16	INT	Link Test	Minor	Major	156	Part-fixed			When there are error/warr triagle shows on the botto Ray Hammond <hammo1r> The following scripts will n displayed to the UI: EBDRS_TUI_LT_004 - EBD PDRE_TUI_LT_005 - Shift T PDRE_TUI_LT_006 - Promp TUI_TS_LT_002 - Commit N TUI_PDRE_LT_002 - Comm TUI-TCE_LT_005 - Calculat TUI-TCE_LT_006 - Allocate Ray Hammond <hammo1r> Retested in release BLD_H The following error manag 156 retest.doc) attached: EBDRS_TUI_LT_004 - EBD PDRE_TUI_LT_005 - Shift T PDRE_TUI_LT_006 - Promp unavailable

Observation Report | Data sheet | Releases | Test Progress | IPAS Environments | Summary

Detailed statistics produced (with a little manual intervention) from the Excel sheet shown immediately preceding this. This shows the number of defects raised and closed each week, and a breakdown of the number at each status at this level of frequency. Most of these were raised by me, and during this period I had a week's holiday. Guess which week!

Q15 =IF(P13>Q13,Q13,Q13-P13)

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	Status			30-Jul	06-Aug	13-Aug	20-Aug	27-Aug	03-Sep	10-Sep	17-Sep	24-Sep	01-Oct	08-Oct	15-Oct	22-Oct	29-Oct
2	Closed		277	36	48	62	100	134	145	145	194	217	227	238	254	267	277
3	Deferred		0	0	0	0	0	0	0	0	4	0	0	0	0	0	0
4	Failed		0	0	0	0	0	0	2	2	2	0	0	0	0	0	0
5	Fixed		0	0	0	0	0	0	1	1	4	5	1	0	0	0	0
6	NOT TUI/TCE		1														1
7	Open		11	9	14	18	21	24	28	28	16	8	21	16	18	15	11
8	Part-fixed		1													1	1
9	Re-opened		0	0	0	0	2	2	3	3	0	0	0	0	0	0	0
10	Rejected		13	0	0	0	3	3	5	5	5	8	8	9	9	13	13
11	To retest in Cardinal		8	0	0	0	0	0	0	0	0	0	5	9	3	7	8
12																	
13	Total		311	45	62	80	126	163	184	184	225	238	262	272	284	303	311
14																	
15	Raised this week			45	17	18	46	37	21	0	41	13	24	10	12	19	8
16	Closed this week			36	12	14	41	34	13	0	49	26	10	12	16	17	10

Defect hints, tips and tricks.

The rest of this paper has many hints, tips and tricks that are useful in enabling us all to write better defects, understand the defect process more fully or handle defects more efficiently. The detailed items are gathered primarily from first-hand experience, and some are specifically related to a company or project where I was working at the time. However, there is a wealth of teaching points to be drawn from these projects and examples, and I hope that you will take each item not as a pointer to 'you must do this', more 'can this help me in my company / project'?

Individual items are listed in no special order, and some are obviously closely related. Grouping by linked items would be difficult, as some topics have a general applicability, whilst others are linked to 3 or 4 others, which are not related to each other. Many of the overall body of items are matters that, on reflection, are common sense. They are merely offered here because I have encountered individuals who have found an item particularly helpful, suggesting that although it is common sense, it is not known to everybody. If this can help you learn a lesson without having to do it the painful way, then it will have been worth the pain I have experienced in getting to this point in my testing career.

Finally, the discussion about defects is more focussed on 'wet' testing, rather than 'dry' testing. Many of the items discussed can equally be applied to reviews, static analysis and other non-execution phases of testing.

Writing better defects gets them fixed

Think back to the categories of 'Good', 'Bad' and 'Poor' defects. We need to aim for the first or second of these categories – not just aim for the first category, as that could mean just raising easy defects. You also may not know if a defect is a good defect until it has been fixed. I have raised many defects that I thought were 'good' defects and they ended up taking much time and effort to resolve, and a significant number that looked horrendous, but were quickly fixed and arrived on my desk-top in the next software release.

The defects that we raise should be readable, and polite. All the necessary detail should be provided (from what you know now) to enable the defect to be picked up, diagnosed and fixed by **ANY** developer and tested by **ANY** tester. Give it a nice snappy title that sums up the perceived problem. Remember, it may not be a problem with the software under test, but a problem with the tester, which is developed in a later section of this paper.

What you write on your defects depends upon any defect tracking procedure that is available where you are working. We are aiming for a clear description of what the perceived problem is. This has to be factual and not personal in any way. The gold standard is a defect that can be resolved correctly without the person undertaking the corrective action needing to come back to you with more information that you could and should have supplied in the first instance. The supplied information should aim for defects that are reproducible in the development environment.

At one work-place, a defect was raised and one of the responses contained expletives – swear words. Here, the customer had access to the defect tracking system, so this was very sensitive area, and the offensive language had to be removed, at a considerable cost. The defect tracking system did not allow any previous text to be amended, so a back-end database amendment was required. For anything you write on a defect, would it stand up to the "what would I think if I received that" test? Ask yourself that before you press the <send> key.

The following can serve as a guideline for the information required

1) What were you doing
2) What happened
3) What do you expect to happen
4) The dataset you were using
5) How did you see that it was incorrect (perhaps an SQL command)
6) Screenshots etc, as necessary

Is the defect repeatable? Rex Black suggests that you undertake the action three times and record whether the same thing (the problem outcome) happens each time. [Black 2003]

Review other people's defects.

If you wish to write better defects, look to those written by others. Why does Fred always get his defects fixed? It could be to do with the quality of his defects, the information that he provides and the phrases that he uses. So, look at others' defects, and ask the star defect-writer to review yours. This is not to 'dumb down' to the lowest level, but more to raise up to the highest standards. The peer-review process will also bring some standardisation on the evidence you provide, and the priority that is assigned. One of the little things that I do is if any evidence files are attached to a defect, these are always additionally held on a project file-share, with the defect id incorporated into the prefix of the filename. So "087_screenshot.doc" is a screenshots associated with defect id 87. That way, evidence is always available, even if the defect tracking spreadsheet/application is not accessible.

Get a reputation as a good defect writer

"Oh good, it is one of George's defects". I once had someone say to me that one of mine was the best defect he had ever seen. It is not a defect I would have held up as a particularly good example; it stated what was wrong, the impact that it had, and a possible solution, using the system data that the defect recipient was the custodian of. In this instance, the information in my defect report enabled the recipient to find and fix the problem quickly and efficiently. Remember, however, that the purpose of writing a defect report is to get something clarified or changed. A defect report is not a candidate for the Nobel Prize for Literature!

Wouldn't it be nice if developers were falling over each other in an effort to be 'the one' – the person that would have the honour of addressing my defect? They haven't even read the defect yet, but they know that because it is one of mine, it will be a good one.

Now there is a thought that should spur us all on to write 'better' defects.

Study defects

This point is not so much about getting better at writing defects, but being able to relate a new defect to an existing open or closed defect. How do you know about open or closed defects unless you study them? There may be long-standing defects that you know something about. Pass on your knowledge to those who currently have the defect on their list or queue. Studying defects gives you a broad idea of what is outstanding, and may help in identifying defects that should be closed, discussed in a later section. It is also by studying defects that you can get ideas of what to test. If there are six defects currently open on report pagination, perhaps the three reports that you are responsible for should have similar tests applied.

These are not only ones that you have raised, but also those raised by colleagues, and even rejected defect reports. By looking at defects that have been raised, it will aid you to be clear, concise and relevant in any defects you raise, it will help you to not raise duplicate defects, and realise situations that seem to be defects but which are not. If there have been 14 solved defects in the main data validation routine in the last 5 weeks that may highlight the need to have an extensive regression test pack available. If a code fix is delivered for a problem reported in "invoice production", and the amended sub-routine is "change surname", you may be a little suspicious.

Studying defects can also mean revisiting defects from former projects, or phases of the same project. If it is useful, or inspiring, study defects, including those raised by others. These can be individuals in different departments or even in different companies. It can be useful to examine 'classic' defects, where here we are **not** concentrating on the problem, but the way it is described. Brett Gonzales has some fine examples, probably due to be published during 2011.

I have a reputation of knowing defect numbers, and some colleagues tease me about this in a gentle way. They describe a defect at me and see if I can give the defect number. Sometimes I am able to perform my party trick, and actually get the number correct. Why do I say this? This comes from a study of defects in my current project (whichever project that is).

As an aside, an observation of mine is that the ICT world in general and software testing in particular is populated by individuals more towards the autistic end of the spectrum. I recognise some of these traits in myself. For a thoroughly inspiring look at the employment of autistic people in the testing industry, see [Sonne 2006]. Please don't misquote me on this; I am not saying that all testers are autistic, or that any particular individual has characteristics in that

direction. My work-based 'party piece' of quoting defect numbers is an indication that I am on the autistic side of the mid-way point.

Raise problematic items as soon as possible

On some projects, there is a tendency to prioritise the testing as follows:

a) Functional
b) Performance
c) Usability

I am sure that I am not the only one to have worked on a project where go-live was sometime after functional testing was completed, but before usability testing was completed. (This was partly because major usability and performance issues were not found until users encountered problems in the first days and weeks after go-live).

If you are involved in FUNCTIONAL testing, you are not primarily looking for performance and usability problems. But if you find one of these you should report it [or get it reported] as soon as you find it. The same goes for other functional errors, those that are not directly "yours" in terms of testing ownership.

i) In order to test your process, you need to create data, using part of the application being tested by Susan. If you encounter a problem, let Susan know, and give her screen prints, etc., so that she can raise a defect as necessary (or indeed ask you to raise it). Of course, she may already know about it.
ii) If a performance issue is apparent, raise it. It will possibly be quite an important performance issue, as often data volumes used in functional testing are nothing like those likely to be encountered in live usage. "Using a small database, I have encountered a twenty seven second wait, between the key press, and the populated screen being displayed". I once worked on a project where, as a functional tester, I was not allowed to raise performance issues. Therefore, I would find a problem, and lead the performance testers to where the problem was. They would take my problem, run some database stats on the SQL query, and provide a VERY thorough investigation of the situation. Another performance problem could only be resolved by whole-scale database table changes; one Oracle table being split into five different tables. This change had a big effect upon functional testing, as some parts of the application that previously worked now encountered serious problems – program crashes. If that problem has been reported earlier, much project angst could have been avoided.
iii) Usability problems. Many usability problems can be around the area of error messages. Functional testing is performed to show that when data is incorrectly entered, a message is displayed. The exact meaning of the message is not necessarily the prime goal of functional testing. The message "A_ABA_AIND: Duplicates not allowed. Please Contact the Help Desk" is not very helpful. It was actually output when the user tried to create a supplier with the same supplier reference as already existed. 'A_ABA_AIND' referred to the ORACLE database constraint that had been violated. A more helpful message would have been "A supplier with this reference already exists. Please amend the reference and try again". What could the Helpdesk have done, that the more helpful message did not?

Spelling mistakes or grammatical errors on warning messages are best reported as and when found. A favourite of mine is "affected" and "effected". If there are 4 or 5 defects raised on the spelling / grammar of error messages, this may give rise to ALL messages being looked at by the

development team, to trap and resolved any remaining difficulties. The information on error and warning messages can have a dramatic effect on whether the user-base will actively seek to use the application, or whether they will find others ways to achieve the same ends.

Another significant usability instance involved a screen with three columns for data entry, and four rows. Two fields on a particular row were Special Duty Start Date, and Special Duty End Date. These could be both blank, both completed, or just the Start completed. The problem was that use of the Tab key moved to the next entry in the same column, before moving onto the next column (at the top). So tabbing from Special Duty Start Date did NOT move to the connected field, Special Duty End Date, as would be expected. This is a very minor problem with the application, with a significant impact on usability.

Encourage developers with their unit tests

Developers write Units Tests for a variety of reasons, but primarily to check that the software performs as expected after changes have been made. Does it still perform as expected in these areas that have not changed, and do the new features match expectations? If you have encountered problems with the delivered software at a previous version, it seems sensible to incorporate detail into the Unit Tests to make sure that the tests you did will not fail in the future as they did in the past. Talk to development colleagues about ways of adding to their Unit Tests, or improving the scope of these tests. If more can be checked by Unit Tests, preferably by automated Unit Tests, then that will improve the quality of the software that is delivered to you. With Continuous Integration and/or Overnight builds, automated Unit tests can be run several times a day. See if you can assist with these; it may be beneficial for all concerned. This is using past defects as a starting point so the more that can be checked BEFORE you receive the code. Previous defects may just help prevent future defects.

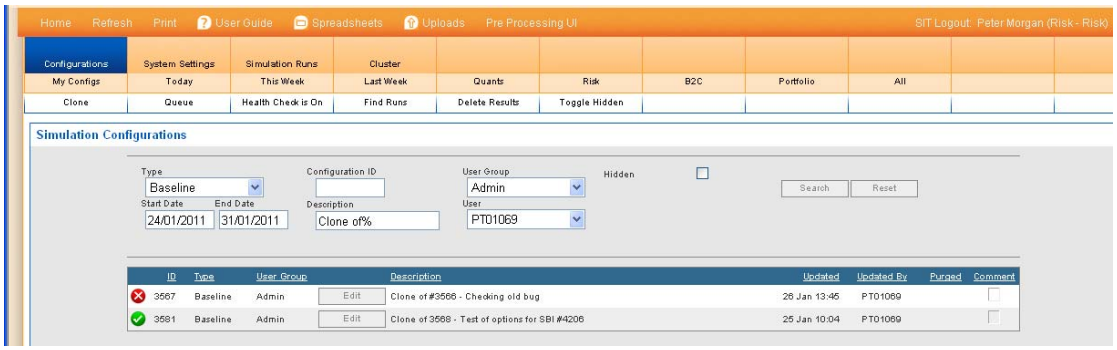
Defect only closed by the originator

This guideline is not one that should never be questioned, but the main point is that developers or project managers should not close defects just because they don't like them, or don't agree that it is a problem. In an ideal world, it would only be originators that closed their own defects. If a defect is raised by a tester, it should be a tester that closes it. Provided that changes have been made (test script, executable code, user manual, etc.), the defect should remain open until the changed items are available in the environment in which the defect was raised. A System tester should not close a defect raised by those (testers or users) involved in UAT

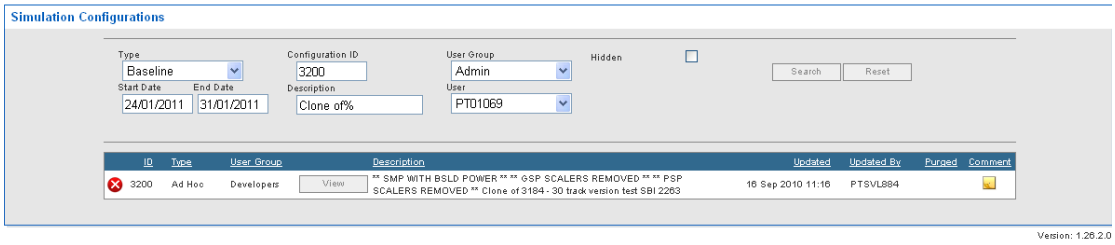
Not everything requires fixing

Not everything you raise requires fixing – you could have made a mistake yourself, or it could be because of a lack of understanding. Even if a change is required, it could be a User Manual, or Installation Instructions, rather than the software you are testing.

Here is an example of something that appears incorrect on first sight. In the first of these screen prints, the middle of the image has the ability to input parameters, and then search using the values supplied. In this instance, a Type, a Date Range, a Description (with a '%' wildcard character), a User Group and a User are specified, and the two instances returned match each of these parameters



Now the Configuration Id is completed, with a value 3200, but all other values of the parameters remains the same. The ‘search’ option is again employed, and one entry is returned, which matches on the Configuration Id, but no other values. Surely this is wrong.



The answer here is that ‘it is what the users want’. The use of a Configuration Id will over-ride all other parameters, and return a maximum of one instance that matches the Configuration Id supplied. [It is a maximum of one instance because Configuration Id is a unique key on the database.] There was nothing to fix, as the software met the user requirements, and the defect (that I raised) was returned to me for closure. In this particular instance, the behaviour was noted for inclusion in the (as-yet-unwritten) User Guide.

Software version and test environment are vital.

In my current place of work, there are three environments that I use for the Sunshine application. These are not all used with the same regularity, and one of the environments is mainly used to investigate difficulties observed by users in their UAT activities. Two additional environments are also available, one of which is the production system. Not all environments will be at the same software release, and environments are usually upgraded at lunchtime when required. So it is very important to record which environment and which software release if a defect is raised. It is possible that a defect reported in the Pre Production or Production environment could already have been fixed and be available in System Test, awaiting deployment.

This means that the version should ideally be available somewhere, whether it is displayed on the User Interface (if there is one) or using one visible on one of the properties of a delivered object. If there is no way of determining the ‘version’, some other mechanism may need to be used; perhaps the date/time stamp. The Sunshine application has both front-end and back-end releases. The front end releases have the version displayed on the UI, but it is less easy to determine the back-end release, so this is down to good documentation. Where a deliverable has a clearly defined and observable version, it is a non-trivial defect if two successive software releases have the same version number. (On some projects I have worked on, having close interplay with the development staff, we arranged that every delivery released beyond System Test had a new version number. Several in between deliveries into System Test could have the same version number).

Versioning does not only apply to delivered executable code, but other parts of the overall delivery; specifications, test plans, test data, etc., etc. At one former place of work, a specification had the version number on the front page in three places. This in itself was not a problem. The REAL problem was that these three occurrences did not match. That is a non-trivial defect.

Give praise (as necessary) to your development colleagues.

Two of the least used words in the English language are ‘thank you’, used together, and in that order. Genuine praise can encourage others to achieve high results. Be careful that the praise is genuine; “It is the turn of somebody in IT to receive the monthly company commendation, and I thought that you would do”. When that happened to a former colleague, it was about as de-motivating as it gets.

It is a sad fact that if you are leaving a job, you will get more thanks in the last 4 weeks when you are working out your notice than you got in the previous 10 years work at the firm. A culture of acknowledging hard work and dedication can make people work better, and encourage them to stay. Or maybe it is different in Belgium

Track your own defects.

I always like to know what is happening to my defects. Have any of them been changed today? If so, what has changed? Someone may have a change in hand for another reason on a module in which there is outstanding defect raised by you. “You have this module booked out to you. Whilst doing your amendment, can you fit in mine please? Think what it will do, if you meet your target, and fix more than you were asked to”. In a polite and friendly way, keep your prize defects in the spotlight. Negotiate, by assisting with unit test ideas in exchange for your current no #1 item being fixed. If all else fails, but people lunch. That can be good for all of you!

Let developers know what you intend to test

With new functionality, it can be useful to let developers know what you intend to test. That way you can actually build in quality, and ‘raise defects without raising defects’, by circulating your test intentions (but not necessarily your test plans)

“When I get this new software, what I intend to do is use the new data extract routine, and I expect that the pre-existing data load routines will be able to load the extracted data, and that the extracted file names will be suitable for uploading. Then I will extract an empty file, and see if this can be loaded again. Finally, I will extract data, reload it and run based on the reloaded data. That should give the same results as I had 2 days ago”. This is a real example, which I used in the last week of January 2011.

I will also use this technique with defects, particularly ones that have failed retest. “In testing defect #1234, I used 7 input files, and expected 6 to load successfully, and file MasterTradeDescription_20110125.xls to fail. However, in addition, file MasterTradeDescription_20110129.xls also failed. All input files are attached.”

If there is a problem, always raise a defect.

Why do we always need to raise a defect? Sometimes, my list of defects can be the basis of a regression pack. If you don't raise it, you may not in the future test for it. That may be fine now, but what happens if you are unexpectedly ill, or hit by the proverbial bus? This guideline applies to emergency releases, perhaps even more than 'standard' releases.

Recently I had 4 deliveries of software in a day, deliveries of the same software that was urgently required in UAT. As the tester in a 3-man Agile team, I was the gate keeper on quality. On the first 3 deliveries, significant problems were encountered, including items that had never been seen before. Each problem was 'logged' (on a note-pad, but more importantly, demonstrated to development colleagues). At this stage, screenshots were saved where necessary.

The important goal was to make delivery to the UAT environment as soon as possible. Formally logging the defects had to wait until the initial target had been met. It was at the formal logging stage that the screenshots previously saved were used. When the delivery to UAT took place, the defects encountered on the testing day had been fixed. The potential use of the defects raised in retrospect is in the generation of automated unit tests and the regression pack.

PRIORITISE defects

Not all companies have the same ways of prioritising defects, but there needs to be some way of determining which items should be investigated and (if necessary, fixed) before others. These remarks about the relative importance of defects and a proposed order of investigation categorises according to the business impact ('severity') and testing impact ('priority'). How many categories of severity have been discussed at length over the years, and although important, is not discussed here. What is important is how each category (e.g. 'showstopper', 'high', 'medium', 'low', etc) is defined, so that it should be possible to clearly determine the values of any individual defect. Everything is clearly not a 'high'. If everything was defined at e.g. the same severity, then there would have to be another way of further evaluating the order items are to be addressed. It is also important to remember that the importance of defects (both the business impact and testing impact) can change over time. What is relatively unimportant today could become critical in 6-months time.

It is possible to have individual defects that have a high severity (business impact) and a low priority (testing impact). An example is a spelling mistake on the home page of a web-site. The appearance of a company name can be difficult, perhaps because the company logo spells a common item in a special way. Company I am currently on contract for is EDF Energy. We can all spell that – but may well spell it wrong. The important thing is that the first letter of the initials is lower-case, whilst the other two are capitals. This is incredibly important for the company image. The name must not only be spelt correctly, but displayed in the correct font, and in the correct colour. Very high severity, yet very low priority, in terms of the impact on the greater testing task.

There is also the converse, a defect that has a high impact on testing at this time, but a low business impact. The business impact could change going into the future, but right now, it is very low. Take a company that has 31st December as its year end. A whole stream of testing is dependent upon being able to perform a year-end closure, and the inability to perform this has a serious impact on testing. Lots of tests cannot be run. Yet in February, the business put little store on the ability to perform year-end closeout. There are many more important things to address. When September comes, the business will be increasingly concerned if the year-end closeout is still incorrect, and will be inventing further severity levels if at mid-November the process is still broken.

Do not be fooled by developers

Sometimes, just sometimes, developers try and blind others by algorithms, and discussions about mandatory database constraints, or something similar. On a former project, Oracle forms was used. In accessing Staff records, an option was available to display holiday details, and if this was selected, another window would be opened with the data displayed. However, if no holiday details had been recorded, a blank window was displayed, this after a message had been displayed “Query returned no records”. The development answer was: “That is the way Oracle forms works”. My answer was firm but polite. “Excuse me, but it **IS** possible to prevent the message being displayed (because other parts of the same product handle similar situations with no message).”

Responses that I will routinely challenge are the following two:

a) “It does not happen on my machine”
b) “But the users will never do that”

In the case of the ‘doesn’t happen on my machine’ reply, request a visit to your terminal, and show the very real difficulty that you are having

If it can go wrong, some clever user will do it. It sounds very trite, but it is not possible to build a fool-proof product because there are some very clever fools around. This is especially true if there is a significant user base – in a previous project I was involved with Post Office automation. This was to put terminals onto every counter in the 19,000 Post Offices throughout the UK. That is a total of 57,000 users, some of whom are very computer literate, and some

being dragged kicking and screaming into using computers. Some of these users did very silly things, but the software allowed them to do it.

This situation is even more likely to happen if the application is an “open access” internet product. What will happen if the input sequence is a four screen transaction, and the user is timed out, or logs out, or losses their connection after 3 of the 4 screens? Some of these would happen to genuine users, through curiosity, mistake, or with users like me, through malice.

On one large project, a defensive programmer catered for an error that the analyst said would never happen. In order to protect herself, the error report said, after some details of the problematic records: “This error should never happen, and as it has Joe Bloggs owes me £5.00”. One night there were 400 of these error reports!

Raise different defects for different processes

If the problem is evident in two or more different processes, at least **THINK** about raising more than one defect. Perhaps it is a failure in surname validation, where the name ‘O’Neil’ is rejected because it has a supposedly illegal character (“apostrophe”). Processes are Insert/amend Customer, Insert/amend Supplier, and Insert/amend staff member. If this is definitely three separate problems (that is surname validation is not separately compiled called module), then three separate incidents should be raised. What you don’t want is to raise one and find not all instances that you know about have been fixed. In cases of doubt, ask development staff if solving the surname problem in Insert/amend Supplier will cure the same problem in the other processes.

Do not raise MANY, MANY problems on the same report

What I detest as a tester hate is having a half-fixed defect. An informal rule on one project I have worked on was “one problem, one incident raised”. I would not always go that far – perhaps a report program with a field label spelt incorrectly, with the wrong number of surrounding space characters, and incorrect capitalisation could all be raised on the one report. However, at the other extreme, I have once received an incident where there were 57 separate parts to the same report. The really annoying part was that some of the issues were my responsibility to solve (responsible for the project standing data), but over half were instantly identified as “not mine”. Some of the other instances on this mammoth incident were later assigned to the correct teams on further investigation. A multipart defect can take a long time to close. We do not want to play the numbers game, but on the other hand, the aim has to be to get defects closed out. Raising individual defects concentrates the attention of the developer on the single matter reported, rather than shelving the multi-part defect because there is never sufficient time to fix all of the items in one morning.

Get closed all defects you can

On most projects there are ‘dead’ defects. In this, I am not talking about defects that have been knowingly fixed, but not yet re-tested. Here, the subject is defects that were raised some time ago, and perhaps the facility to create the particular data combination no longer exists. There has been no fix, but the facility to create a customer with no surname has been removed, or the

problem with screen FGL-03A is not evident as this screen has since been removed. The defect is therefore no longer appropriate. Other situations are where a defect has been unintentionally fixed, and the defect is not closed, because no-one is aware that it has been fixed. Open defects that are 'dead' cost the project time and money (if only by those that look at the defect list, and 'automatically' discount some defects, because they know nothing about it). In this, I am not advocating a return to 'test all outstanding defects on any release', but if you can assist in closing 'dead' defects, it saves time and money for the whole project. Finding those that are possible 'dead' defects comes from knowing what defects are not closed – back to studying defects discussed earlier.

My experience is that if there are 200 open (or perhaps it is better to say 'not closed') defects, at least 10% will be wrongly open. I get the figures of '200' and 10% through studying defects, although it on some projects or in some companies where I have worked in has been considerable higher.

There is also the psychological benefit of having less defects requiring action.

Don't play defect ping-pong

A defect report is not the place to have a long drawn-out design discussion, or an argument about the cause of a problem. When I raise a defect, I don't want to know who is NOT responsible for fixing it. I have found the problem and having raised it and got people to agree that it is a problem, the next thing I wish to see (both in terms of the software and the defect report) is a fix delivered. As a tester, perhaps I can facilitate in getting people to talk. My ideal solution on 'hard defects' (where all agree that it is a problem) would be to get those together in a lockable room, with me and the only key on the outside. The only way to get the door unlocked is to find someone who will be responsible for isolating the problem, identifying the cause, and building a robust, testable solution, in a timely manner. If they won't talk of their own will, a locked door with me on the outside can be a way forward.

The aim has to be to get defects solved; not just moved to someone else. Every time a defect is assigned to a new person, or a new team, it will cost the project 2 hours – at least 2 hours. Therefore, think before just shuffling off defects to someone else. Otherwise you have just added some more multiples of 2 hours. A quick 5 minute chat could save a wasted 2 hours, for somebody on the project.

If you can "feel" a problem, keep going until you find it.

This is probably the hardest of the tips to explain, or illustrate by example. Some colleagues can "feel" a problem, or "smell" it. We need to learn to know when to trust that gut feeling. How good is your gut feeling? Is it trust-worthy? I would say that my feelings of unease are accurate 70% - 80% of the time. So, in persevering in search of a problem, remember that you could be wrong. Also, only trust your gut feeling for problems. Do NOT trust your gut feeling if the sense is that everything is OK. If you go by your instincts in this last-mentioned situation, then testing could end prematurely.

Any examples of instances where a tester has an uneasy feeling are of course very subjective. Here are two examples:

An on-line transaction causes a background job to be initiated. If a week's worth of data is processed, this takes 12 minutes to complete. The time taken for 6 months data to be processed is also 12 minutes. Perhaps something is not quite right here ……

Four input screens, allowing data to be entered / amended for Customers, Suppliers, Supervisors, and Managers. These screens have a different number of fields, with an incomplete cross-match of the data required [date of birth on some of the four screens, but not all, etc.]. There is one common report, that prints this information, for the four types of data entered, and all of the four types of data give rise to the same report layout, and the same fields output. Perhaps something is not quite right here ……

The trading power of defects.

On any project, there is a relative value of defects of different severities. Sometimes, this is written in the exit criteria, but not always. Let us take a project that has 5 values for severity; critical, high, medium, low and trivial. The stage-end of System Test states that there should be **no more** than the following, to exit this stage

i)	1 Critical	or
ii)	10 High	or
iii)	50 Medium	or
iv)	150 Low	

There is already an explicit relative value of defects, and this value should be used by all, if it ever comes down to the numbers game, in an attempt to reach the exit criteria. From the exit criteria, 1 Critical = 10 High, 1 High = 5 Medium, etc. Remember, it is '1 Critical OR 10 High ……..'. Therefore, if you had 151 Low defects, the stage end would not have met its exit criteria, 1 Critical and 1 Low fails the criteria, as this equates to 151 Lows. But so would 6 High, 38 Medium and 96 Lows ……

There is also another category of defects that has not been classified in the exit criteria. The number of 'Trivial' defects is undefined. In situations like this, it is useful to request clarification. If everything that is recorded is a genuine defect, I would venture to suggest that 500 trivial defects would make the system absolutely un-usable. Finally, not all spelling mistakes are classed as trivial defects. A missing 'NOT' on a screen log-on message could be a critical defect, as could the misspelling of a company's name on a new web-site under construction.

Does your project have a written down relative value of defect severity? If not, then try to get consensus on what this should be, and move towards using this.

The idea for the trading power of defects was inspired by the Sunday School reward system, well described in [Twain 1876]

Talk talk talk

There is great value in talking to those around us about our actual or intended defects. This is not just other testers, but developers, business users or those who receive print-outs from the software you are testing, for example. In an e-mail world, it can be very easy to fire off a quick response, rather than picking up the phone, or walking over to the desk of the person involved. At a place where I once worked, there were 2 buildings, 80 metres apart. I was always getting up from my desk to walk those 80 metres. Face to face contact achieves a lot. The bonus for me was to have 2 minutes of thinking time. The exercise was also good to keep the waist-line in check.

Face to face meetings will not always work, particularly with geographically and time-zone removed working communities. The technology is there to overcome these difficulties, so use all means [conference calls, video links, webinars, etcetera], and find ways of talking.

Summary

In the foregoing 'Defects hints, tips and tricks' section, there are many suggestion to help and assist. Some of these will not be appropriate for your company or your project. However, there are other items that you can and should be used, either broadly unchanged, or adapted for your specific work place. Understanding the defect process and writing better defects will get defects fixed quicker. It could just turn a mediocre product into a good one, meaning success for the company and ultimately for the whole software development team. Developers and Testers.

So, if there are thoughts and suggestions that may work for you, go and try them. You know it makes sense

References

Black 2003 – Rex Black: Critical Testing Processes
Carroll 1871 – Lewis Carroll (real name Charles Lutwidge Dodgeson): Through the looking-Glass and What Alice Found There
Morgan 2002 – London SIGiST: Tips for Testers
Morgan 2007 – London SIGiST: Testing Tips
Sonne 2006 – Torkil Sonne, EuroSTAR 2006: A special member for the Dream Team
Twain 1876 – Mart Twain (real name Samuel Langhorne Clemens): The Adventures of Tom Sawyer