Kelly Whitmill

Kelly Whitmill has over 18 years experience in software testing. Most of that time his role has been that of a team lead with responsibility for finding and implementing effective methods and tools to accomplish the required tests. He is particularly interested in practical approaches that can be effective in environments with limited resources. He has a strong interest in test automation. He has worked in both small and large company environments. He has worked on PC-based, Unix-based, and Mainframe-based projects. He currently works for the IBM Printing Systems Division in Boulder, Colorado.

Writing Effective Defect Reports

Kelly Whitmill
IBM Printing Systems Division
6300 Diagonal Highway
003G
Boulder, Colorado 80301
(303) 924-9145
whitmill@us.ibm.com

Writing Effective Defect Reports

Introduction

Defect reports are among the most important deliverables to come out of test. They are as important as the test plan and will have more impact on the quality of the product than most other deliverables from test. It is worth the effort to learn how to write effective defect reports. Effective defect reports will:

- Reduce the number of defects returned from development
- Improve the speed of getting defect fixes
- Improve the credibility of test
- Enhance teamwork between test and development

Why do some testers get a much better response from development than others? Part of the answer lies in the defect report. Following a few simple rules can smooth the way for a much more productive environment. The objective is not to write the perfect defect report, but to write an effective defect report that conveys the proper message, gets the job done, and simplifies the process for everyone.

This paper focuses on two aspects of defect reports, 1) the remarks or description and 2) the abstract. First, lets take a look at the essentials for writing effective remarks.

Defect Remarks

Here are some key points to make sure the next defect report you write is an effective one.

1. **C**ondense - Say it clearly but briefly
2. **A**ccurate - Is it a defect or could it be user error, misunderstanding, etc.?
3. **N**eutralize - Just the facts. No zingers. No humor. No emotion.
4. **P**recise - Explicitly, what is the problem?
5. **I**solate - What has been done to isolate the problem?
6. **G**eneralize - What has been done to understand how general the problem is?
7. **R**e-create - What are the essentials in triggering/re-creating this problem? (environment, steps, conditions)
8. **I**mpact - What is the impact to the customer? What is the impact to test? Sell the defect.
9. **D**ebug - What does development need to make it easier to debug? (traces, dumps, logs, immediate access, etc.)
10. **E**vidence - What documentation will prove the existence of the error?

It is not just good technical writing skills that leads to effective defect reports. It is more important to make sure that you have asked and answered the right questions. It is key to make sure that you have covered the essential items that will be of most benefit to the intended audience of the defect report.

Essentials for Effective Defect Remarks

Condense

Say it clearly but briefly. First, eliminate unnecessary wordiness. Second, don't add in extraneous information. It is important that you include all relevant information, but make sure that the information is relevant. In situations where it is unclear how to reproduce the problem or the understanding of the problem is vague for whatever reason you will probably need to capture more information. Keep in mind that irrelevant information can be just as problematic as too little relevant information.

Condense Example	Defect Remark
Don't: Suffers from TMI (Too Much Information), most of which is not helpful.	I was setting up a test whose real intent was to detect memory errors. In the process I noticed a new GUI field that I was not familiar with. I decided to exercise the new field. I tried many boundary and error conditions that worked just fine. Finally, I cleared the field of any data and attempted to advance to the next screen, then the program abended. Several retries revealed that anytime there is not any data for the "product description" field you cannot advance to the next screen or even exit or cancel without abending.
Do:	The "exit", "next", and "cancel" functions for the "Product Information" screen abends when the "product description" field is empty or blank.

Accurate

Make sure that what you are reporting is really a bug. You can lose credibility very quickly if you get a reputation of reporting problems that turn out to be setup problems, user errors, or misunderstandings of the product. Before you write up the problem, make sure that you have done your homework. Before writing up the problem consider:

- Is there something in the setup that could have caused this? For example, are the correct versions installed and all dependencies met? Did you use the correct login, security, command/task sequence and so fourth?
- Could an incomplete cleanup, incomplete results, or manual interventions from a previous test cause this?
- Could this be the result of a network or some other environmental problem?
- Do you really understand how this is supposed to work?

There are always numerous influences that can affect the outcome of a test. Make sure that you understand what these influences are and consider their role in the perceived bug you are reporting. This is one area that quickly separates the experienced tester from the novice. If you are unsure about the validity of the problem it may be wise to consult with an experienced tester or developer prior to writing up the problem.

As a rule of thumb it helps to remember the adage that "it is a sin to over report, but it is a crime to under report." Don't be afraid to write up problems. Do your best to ensure that they are valid problems. When you discover that you have opened a problem and it turns out to be an incorrectly reported problem, make sure that you learn from it.

Neutralize

State the problem objectively. Don't try to use humor and don't use emotionally charged zingers. What you think is funny when you write the defect may not be interpreted as funny by a developer who is working overtime and is stressed by deadlines. Using emotionally charged statements doesn't do anything for fixing the problem. Emotional statements just create barriers to communication and teamwork. Even if the developers doubted you and returned your previous defect and now you have proof that you are correct and they are wrong, just state the problem and the additional information that will be helpful to the developer. In the long run this added bit of professionalism will gain you respect and credibility. Read over your problem description before submitting it and remove or restate those comments that could be interpreted as being negative towards a person.

Neutralize Example: This example is a response to a developer returning a defect for more information and requesting more details on what values caused the problem.	Defect Remark
Don't: The first clause will probably be interpreted as a jab at the developer and adds no useful information.	As could have been determined from the original defect with very little effort, function ABC does indeed abend with any negative value as input.
Do:	Function ABC abends with any negative value. Examples of some values tested include -1, -36, -32767.

Precise

The person reading the problem description should not have to be a detective to determine what the problem is. Right up front in the description, describe exactly what you perceive the problem to be. Some descriptions detail a series of actions and results. For example, "I hit the enter key and action A happened. Then I hit the back arrow and action B happened. Then I entered the "xyz" command and action C happened." The reader may not know if you think all three resulting actions were incorrect, or which one, if any is incorrect. In all cases, but especially if the description is long, you need to summarize the problem(s) at the beginning of the description. Don't depend on an abstract in a different field of the defect report to be available or used by everyone who reads the problem description. Don't assume that others will draw the same conclusions that you do. Your goal is not to write a description that is possible to understand, but to write a description that cannot be misunderstood. The only way to make that happen is to explicitly and precisely describe the problem rather than just giving a description of what happened.

Precise Example	Defect Remark
Don't: In this example, it is hard to tell if the problem is 1) the twinax port not timing out or 2) the printer not returning to ready or 3) the message on the op panel.	Issuing a cancel print when job is in PRT state (job is already in the printer and AS/400 is waiting to receive print complete from printer) causes the Twinax port to not time out. The printer never returns to a READY state and indefinitely displays "PRINTING IPDS FROM TRAY1" in the op-panel.
Do: Precede the description with a short summary of exactly what you perceive the problem to be.	**Canceling a job while it is printing causes the printer to hang**. Issuing a cancel print when job is in PRT state (job is already in the printer and AS/400 is waiting to receive print complete from printer) causes the Twinax port to not time out. The printer never returns to a READY state and indefinitely displays "PRINTING IPDS FROM TRAY1" in the op-panel.

Isolate

Each organization has its own philosophy and expectations on how much the tester is required to isolate the problem. Regardless of what is required, a tester should always invest some reasonable amount of effort into isolating the problem. Consider the following when isolating problems.

- Try to find the shortest, simplest set of the steps required to reproduce the problem. That usually goes a long way towards isolating the problem.
- Ask yourself if anything external to the specific code being tested contributed to the problem. For example, if you experience a hang or delay, could it have been due to a network problem? If you are doing end-to-end testing can you tell which component along the way had the failure? Are there some things you could do to help narrow down which component had the failure?
- If your test has multiple input conditions, vary the inputs until you can find which one with which values triggered the problem.

In the problem description, to the extent possible, describe the exact inputs used. For example, if you found a problem while printing a Postscript document, even if you think the problem occurs with any Postscript document, specify the exact document that you used to find the problem.

Your ability to isolate, in large part, defines your value-add as a tester. Effective isolation saves everyone along the line a great deal of time. It also saves you a lot of time when you have to verify a fix.

Generalize

Often times, the developers will fix exactly what you report, without even realizing the problem is a more general problem that needs a more general fix. For example, I may report that my word processor

"save file" function failed and the word processor abended when I tried to save the file "myfile". A little more investigation may have revealed that this same failure occurs anytime I save a zero length file. Perhaps, on this release it abends on every save to a remote disk, a read only disk, and so forth. To already know this when you write the report will save the developer a lot of time and enhance the possibility of a better fix to handle the general case.
When you detect a problem, take reasonable steps to determine if it is more general than is immediately obvious.

Generalize Example	Defect Remark
Don't:	Error message for "file not found" error has garbage characters for the file name.
Do:	Error message for "file not found" error has garbage characters for the file name. Every message I tried that expected data to be inserted in the message had the same problem. Messages without inserts were okay.

Re-create
Some bugs are easy to re-create and some are not. If you can re-create the bug you should explain exactly what is required to do the re-create. You should list all the steps, include the exact syntax, file names, sequences that you used to encounter or re-create the problem. If you believe that the problem will happen with any file, any sequence, etc. then mention that but still provide an explicit example that can be used to do the re-create. If in your effort to verify that the bug is re-creatable you find a shorter and reliable means of re-creating, document the shortest, easiest means of re-creation.

If you cannot re-create the problem or if you suspect that you may not be able to re-create the problem gather all the relevant information that you can that may provide useful information to the person who has to try and fix the problem. This may be a time when you consider asking a developer if they want to examine the system while it is still in the problem state or if there is any particular information that should be captured before cleaning up the problem state and restoring the system. Don't assume that it can be re-created if you haven't verified that it can be re-created. If you cannot or have not re-created the problem it is important to note that in the defect remarks.

Impact
What is the impact if the bug were to surface in the customer environment? The impact of some bugs is self-evident. For example," entire system crashes when I hit the enter key." Some bugs are not so obvious. For example, you may discover a typo on a window. This may seem very minor, even trivial unless you point out that every time someone uses your product this is the first thing they see and the typo results in an offensive word. In this case, even though it is just a typo it may be something that absolutely must be fixed prior to shipping the product. Make your best judgment. If you think it is possible that this defect will not get sufficient priority then state the potential impact and sell the defect. Don't oversell, but make sure the readers of the defect have an accurate understanding of the probable impact on the customer.

Debug
What will the developer need to be able to debug this problem? Are there traces, dumps, logs, and so forth that should be captured and made available with this defect report? Document what has been captured and how it can be accessed.

Evidence
What exists that will prove the existence of the error? Have you provided both the expected results and the actual results? Is there documentation that supports your expected results? Since you are writing a problem report it is obvious that you believe there is a problem. Provide anything you can that will convince others also that this is indeed a valid problem. Evidence may take the form of documentation from user guides, specifications, requirements, and designs. It may be past comments from customers, de-facto standards from competing products, or results from previous versions of the product. Don't assume everyone sees things the same way you do. Don't expect people to read between the lines and draw the same conclusions as you. Don't assume that 3 weeks from now you will remember why you thought this was a bug. Think about what it is that convinced you that this is a bug and include that in the report. You will have to provide even more evidence if you think there is a chance that this situation may not be readily accepted by all as a valid bug.

Mental Checklist
You won't be able to go back and study this paper each time you write a defect report. It is important that you develop an easily accessible mental checklist that you go over in your mind each time you write a defect report. Inspections have proven to be the least expensive and most effective means of improving software quality. It stands to reason, that the least expensive most effective means of improving the quality of your defect reports is an inspection, even if it is an informal self-inspection. It is important that using whatever memory techniques work for you that these checklist items get implanted into your memory. In most cases, inadequate defect reports are not due to an inability to write a good report. Usually, we just didn't think about and answer the right questions. This mental checklist takes us through the process of thinking about and answering the right questions.

You may find it useful to apply a mnemonic to the checklist. If you look at the first letter of each item on the checklist it spells CAN PIG RIDE? This is just short enough and obnoxious enough that hopefully it will stick with you. If you spend about 20-30 minutes using this phrase and associating it with the defect inspection checklist, you will probably have that mental checklist implanted in your memory. If ten items are too much to remember, then concentrate on PIG. If you do a good job on these three items, Precise, Isolate, and Generalize it will guide you to adequate and more effective defect reports in most cases.

Template
A defect remark template can prove useful in making sure that the remarks provide the correct information and answer the right questions. Some defect tracking tools may allow a template to

automatically be displayed whenever it prompts for defect remarks. Otherwise, you may have to use cut and paste to insert a template into your remarks. A sample template follows.

Product Details:

Product Name and Number:	
Version, Revision, build and disk number:	

System Details:

Computer Type: PC model, mainframe type, OS Level, etc.	
Memory:	
Disk Space:	
Peripherals attached and used:	
Network connectivity:	
Configuration Details:	

Problem Summary:

Problem Description: (include expected and actual results)

Is this reproducible?

Steps and conditions to reproduce:

Has this problem been isolated?
Has this problem been generalized?
Additional Debug Information: (How to access logs, dumps, etc.)

In effective defect reporting, as in many situations, it is not a matter of if you got the answers correct but more a matter of did you answer the correct questions? These ten points:

- **C**ondense
- **A**ccurate
- **N**eutralize
- **P**recise
- **I**solate
- **G**eneralize
- **R**e-create
- **I**mpact
- **D**ebug
- **E**vidence

Provide a quick checklist to ensure that your defect reports answer the right questions that will be of most benefit to your organization.

Defect Abstracts

The short one line abstract that gets associated with most defects is a very powerful communication tool. Often times, the abstract is the only portion of the defect that gets read by the decision-makers. It is the abstract, not the full description, that gets included in reports. It is the abstract that the project managers, screeners, team leads and other managers look at when trying to understand the defects associated with the product.

The abstract must be concise and descriptive and convey an accurate message. The abstract is usually very limited in length. Because of the space limitations, abbreviations are okay and short accurate messages take priority over good grammar. A good use of key words is essential since many searches are based on the abstract. Keywords such as abend, hang, typo and so forth are both descriptive and prove useful as search words. Where space permits it is helpful to mention the environment, the impact, and any of the who, what, when, where, why questions that you can address in such a short space.

Some defect tracking tools provide default abstracts by using the first line of the problem description or similar defaulting mechanisms. Never take the default abstract. Be as specific as possible. For example, the following abstract is true but doesn't provide nearly as much information as it could.

Abstract: Problems found when saving and restoring data member.

Perhaps a more descriptive abstract would be:

Abstract: xyz's save/restore of data member on WinNT fails, data corrupted

You can never get everything you want in an abstract. Here is a list of items and tips that you try to include in an abstract.

Abstract Checklist

Mandatory:

1. Concisely, explicitly state what the problem is. (not just that there is a problem)

Recommended (space permitting):

1. Use meaningful keywords
2. State environment and impact
3. Answer who, what, when, where, why, and how
4. Okay to use abbreviations
5. Grammar is secondary over conveying the message
6. Don't use defaults

Summary

Testers spend a significant amount of time seeking out and discovering software problems. Once detected, it greatly enhances productivity to report the defect in such a way as to increase the likelihood of getting the problem fixed with the least amount of effort. Making sure that the proper information is provided is more important than superior writing skills. The 10 topics described in this paper

- **C**ondense
- **A**ccurate
- **N**eutralize
- **P**recise
- **I**solate
- **G**eneralize
- **R**e-create
- **I**mpact
- **D**ebug
- **E**vidence

will go a long way toward help you provide the right information in every defect report.

Not everyone reads the entire defect report. Many decision-makers rely on the one-line defect abstract to base their decisions on. It is important to write abstracts that accurately convey the right message about the abstract.

The better you are at writing defect reports and abstracts, the more likely it is that the problems will actually get fixed and in a more timely manner. Your credibility and value-add to the business will increase as developers, managers, and other testers are better able to do their jobs because your defect reports are will written and reliable.

Appendix A

Condense (say it clearly but briefly)

Accurate (Is it really a defect? Could it be user error, setup problem etc.?)

Neutralize (Just the facts, no zingers, no humor, no emotion)

Precise (Explicitly what is the problem?)

Isolate (What has been done to isolate the problem?)

Generalize (What has been done to understand how general the problem is?)

Re-create (What are the essentials in creating/triggering this problem?)

Impact (What is the impact if the bug were to surface in customer env.?)

Debug (What does the developer need to debug this?)

Evidence (What will prove the existence of the error? documentation?)

References:

Rex Black, *The Fine Art of Writing a Good Bug Report*, http://www.rexblack.consulting.com