The Bug Reporting Processes

Rex Black: President and Principal Consultant
RBCS, Inc., www.rexblackconsulting.com

Introduction

Last time, I talked about an internal test process, managing test execution. This is a process that is only indirectly visible to the rest of the development team, in that, as long as you get through all your planned tests, effectively respond to change, and report your findings intelligibly, then for all most people care your testing could occur through voodoo, augury, and a Ouija board.

However, there is an internal test process that produces results that are very visible: the bug reporting process. In a sense, bug reports are the only tangible product of the testing process itself, solid technical documentation that describes a failure mode in the system under test that could affect the customer's experience of quality. Dorothy Graham and Mark Fewster have written that the purpose of testing is to give increased confidence in those areas of the product that work and to document issues with those areas of the product that do not work.[1] This documentation gives management the information they need to decide the priority of the bug and, for those bugs important enough to fix, these reports provide the developer with the information she needs to fix it.

This brings us to why the bug reporting process is so critical. First and foremost, bug reporting is critical because bug reports are the vehicle by which we as testers influence increased product quality. Second, bug reporting is something that we do every day—and many times a day, usually—during test execution phase. Third, bug reports are often highly visible, being read by developers and senior managers alike. The successful test manager must master this process, and help her test team master it as well, in order for the test team to fully realize their potential contribution to the development effort.

Definitions

While the issues of "what constitutes a bug" and "what is a bug report" seem obvious from a high level, as a practical matter a number of semantic problems arise. To start, I define a bug as a problem present in the system under test that causes it to fail to meet a reasonable user's expectations of behavior and quality. Straightforward, right? Well, one person's reasonable expectations can be another person's wild-eyed pipe dreams. When standards of quality or expectation of behavior are neither obvious nor documented in requirements, specifications, business rules, or use cases, my experience is that the reasonableness of an expectation is usually decided by iterative consensus—i.e., discussion between the various concerned managers and staff—or by management fiat.

Having mentioned quality, what can we say about that elusive trait? J. M. Juran, one of the fathers of the quality movement, defines quality as, "Features…decisive as to product performance and as to 'product satisfaction'…[and] freedom from deficiencies…[that] result in complaints, claims, returns, rework and other damage. Those collectively are forms of 'product *dis*satisfaction.'"[2]

So, I can rephrase my definition to say that a bug exists when the product passively fails to satisfy the customer or actively causes customer dissatisfaction. More specifically, a bug is either a defect in the system under test that leads to quality-reducing behaviors or the absence of support in the system under test for quality-increasing features (that a reasonable user would expect). I want to be specific here because one common objection to some bug reports is along the line of, "that's not a bug, it's

---

1 *Software Test Automation*. Mark Fewster and Dorothy Graham. Addison-Wesley, New York, 1999.

2 Juran on Planning for Quality. J. M. Juran. The Free Press, New York, 1988.

an enhancement request." By my definition, the question of whether a quality issue should be resolved in the current release or some subsequent release is a question of business priority for executive management, but it doesn't affect whether a quality issue exists.

A further source of confusion about bug reports arises because the word "bug" is not ubiquitous. Some people prefer "fault," "error," "glitch," or "defect."—among other words. The symptom of a bug is sometimes called the "failure," "failure mode," "event," "issue," "anomaly," or even "apple."[3] I recognize these terms as synonymous, but I like the word "bug" because of its quaint historical connotations.

This brings us to the topic of this article, the bug report, which I define as a technical document written to describe the symptoms of a bug for the purposes of: 1) communicating the impact and circumstances of a quality problem; 2) prioritizing the bug for repair; and, 3) helping the developer locate the underlying defect and fix it.

The Bug Reporting Process

How can we write the best possible bug reports? The following outlines the bug reporting process I use on my projects.

1. **Structure.** Good bug reporting begins with solid, organized testing. Testing may be manual or automated, scripted or exploratory, but it must be something other than just hacking on the product in an aimless attempt to break it. Sloppy testing yields sloppy bug reports.
2. **Reproduce.** Bugs almost always take us by surprise. If we could predict in advance when we would find a bug, we would hardly need to test the product. Reproducing the problem sharpens our understanding of the issue and allows us to document a crisp set of steps that will recreate the failure. I like to use three tries as a rule of thumb. Of course, some bugs are intermittent. In this case, I still write the report, but I note the intermittence; e.g., "This failure occurred two out of three tries." This has the substantial political side-benefit of addressing the issue of irreproducibility head-on, which I'll revisit later.
3. **Isolate.** Many factors can affect the behavior of software, and some of those factors will affect bugs. The way in which a change in the test environment manifests itself in terms of bug behavior is often significant. By changing some key variables, one at a time, in an attempt to change the behavior of the bug, you can write more insightful bug reports. Isolating a bug is not easy; you must understand to some extent how the system under test works and you must think through your tests before you run them. You also must take care not to voyage into the land of debugging (more about this later) or expend inappropriate amounts of time on trivial issues. However, good bug isolation builds tester credibility and gives the developer a head start on debugging.
4. **Generalize.** Often times, the failure mode we observe when we first find a bug is not the most general case. For example, I recently found a problem where a program wouldn't import a specific worksheet from an Excel file. When I dug a bit deeper, though, I realized that it wouldn't import *any* worksheet from any Excel file if the name of the worksheet contained parentheses. It's important, when looking for the general case, not to generalize to the point of absurdity. Not all program crashes, for example, come from the same bug. Nevertheless, trying to find the general rule for how the bug occurs deepens your understanding and helps convince developers and managers that the problem is not an isolated case.

---

[3] The story came to me over the Internet in a test discussion group. In an attempt to defuse the rancor that had arisen in discussing "bugs" the management team renamed them "apples."

5. **Compare.** Testing often covers the same ground, both when a tester runs the same tests against a later version of software and when a tester runs tests that cover similar conditions. These results can provide important clues to a developer when you connect the dots. Is the failure a regression? Does the same feature work in other parts of the product? While this kind of research is not always possible—e.g., when a test was blocked against a previous version—it can save the developers a lot of time if you can find this information.
6. **Summarize.** At this point in the process, we probably understand the failure well enough to answer the all-important question: "How will this problem affect the customers?" To translate that to manager-speak, that is, "Why should I waste my time listening to you talk about this bug?" I mentioned manager-speak because much of the reason we want to write a summary for our bug reports is to communicate in a single sentence the essence and significance of the problem we have found to managers. This summary is invaluable when it comes time to prioritize the bugs. It also gives the bug report a "handle" or name for developers. "Hey, have you fixed that font-change bug yet?" is a question you might ask a developer which is far more meaningful that, "How's work on bug 1716 coming?" Writing a good summary is harder than it looks. Thomas Jefferson once concluded a long letter with the sentence, "I would have written a shorter letter but I didn't have time." Spend the time to write good summaries, because it's the most important sentence in the bug report.
7. **Condense.** Speaking of writing a shorter letter, how about trimming some unnecessary words from our bug reports? I once read a bug report that had nothing to do with computer video games that happened to mention that the tester had just set the high score in Tetris, an 80's video game. Did I care? No. Did I need to spend time reading about that? No. Did the tester who wrote that report make me want to pay close attention to his subsequent reports? No. The best bug reports are neither cryptic commentary nor droning, endless, pointless documents. Use just the words you need and describe only the steps and information that actually pertain to the bug being reported.
8. **Disambiguate.** To disambiguate is to remove confusing or misleading phrases that might make a reader wonder what you mean; e.g., I could disambiguate this article bug using the alternative phrase "be clear" instead of disambiguate. All kidding aside, the idea if for the bug report to lead the developer by the hand to the bug. Clarity is key.
9. **Neutralize.** As the bearers of bad news, we need to express ourselves calmly and impartially in bug reports. Attacking the developer, criticizing the underlying error in the code, or using humor or sarcasm to make a point generally back-fires. We should be gentle and fair-minded in our words and implications, and confine our bug reports to statements of fact, not conjectures or hyperbole. As Cem Kaner has pointed out, too, you never know who's going to read your bug reports—plaintiff's counsel, perhaps?—so to avoid unnecessary appearances as a witness for the prosecution of your employer, you should write only exactly what you mean.[4]
10. **Review.** As I mentioned earlier, a bug report is a technical document. One thing we've learned about quality assurance is that peer reviews, inspections, walkthroughs, and the like are powerful tools for early detection of defects. The same applies to bug reports. Before you dismiss this step as too time-consuming or a needless roadblock to getting bug reports filed as quickly as possible, consider the delays and frustration that can result when a developer wastes time trying to reproduce a bug that was actually a test problem or when a developer can't figure out what a bug report is about and so returns it as "unreproducible."

---

4 *Testing Computer Software*, Second Edition. Cem Kaner, Jack Falk, and Hung Nguyen. International Thompson Computer Press, Boston, 1993.

Let me point out two facts about this process before we go on. First, it's important to remember that writing is creative: two good bug reports on one problem can differ in style and content (but not substance). The purpose of my bug reporting process is not to "standardize" bug reports, but rather to ensure quality. Think of this process as mixing paint or preparing a canvas for a portrait, rather than a "paint-by-numbers" diagram. Second, this process work best when thought of as a checklist rather than a strict sequential process. The first and last steps—structured testing and a peer review—are logical bookends, but if you feel more comfortable reviewing your wording for neutrality throughout, then go right ahead.

In the next section, I'll work through a hypothetical case study illustrating this process. I'll look at a sequence of failure descriptions that one might write to capture the essence of a bug, then I'll cover a few other data you may want to capture in your bug reports. After this, I'll propose some quality indicators for the bug reporting process both in terms of the process itself and the results—the bug reports—that come from it.

A Hypothetical Case Study

To illustrate our bug reporting process, we can continue on with the same case study as before. Recall that you are the test manager for release 1.1 of a Web-based word processing program called Speedy Writer. Under a waterfall development model, the project entered the System Test phase last week, and the System Administration group just installed the second build on your network. Your team has just started the second cycle of testing. Let's look a gradually refined bug report that L. T. Wong, our manual test engineer, writes during this cycle.

L.T. Wong is running an edit functionality test. She creates a new file, inserts some text, then tries to change the font of the text by selecting it, pulling down the font menu, and selecting the Arial font. Suddenly, the screen displays a bunch of symbols, questions marks, and other meaningless noise where nice normal English sentences once were. L.T. starts her bug report and notes:

> *Nasty bug trashed contents of new file that I created by formatting some text in Arial font, wasting my time.*

This might be the way you would describe this problem in a hallway conversation, but how could a developer use this report to improve the quality of the SpeedyWriter product? With an eye towards helping the lucky developer understand what's going on, suppose that L.T. goes back and reproduces the problem a couple more times, refining her thinking about what went wrong and how to make it happen again. Subsequently, she might write the bug report as follows:

> *Steps to Reproduce*
>
> *1. I started the SpeedyWriter editor, then I created a new file.*
>
> *2. I then typed in four lines of text, repeating "The quick fox jumps over the lazy brown dog" each time, using different effects each time, bold, italic, strikethrough, and underline.*
>
> *3. I highlighted the text, then pulled down the font menu, and selected Arial.*
>
> *4. This nasty bug trashed all the text into meaningless garbage, wasting the user's time.*
>
> *5. I was able to reproduce this problem three out of three tries.*

This is much better. A developer can read this and get a head-start on debugging the problem. However, remember that the SpeedyWriter product, as with any other application, supports a wide

range of features and configurations that may affect the behavior of this bug. Perhaps we should see what happens when we try to isolate the key variables that affect the bug? L.T. tries a few experiments and adds the following section to the bottom of her bug report:

> *Isolation*
>
> *On the vague suspicion that this was just a formatting problem, I saved the file, closed SpeedyWriter and reopened the file. The garbage remained.*
>
> *If you save the file before Arializing the contents, the bug does not occur.*
>
> *The bug does not occur with existing files.*
>
> *This only happens under Windows 98.*

This is a very specific failure mode, so L.T. decides to try a few other fonts, as well as checking to see if the font effects and styles have anything to do with the bug. She finds that both Wingdings and Symbol fonts suffer from the same problem, while the bold, italics, and underlining have no effect on the bug's manifestation. She deletes the potentially misleading reference to font styles from step two of the report and adds the following line to the Isolation section:

> *Also happens with Wingdings and Symbol fonts.*

Next, L.T. reviews the test logs from the previous cycle of System Test as well as looking at some test reports from the developer's Component Test and Integration Test phases. She finds that this particular feature has been tested quite a bit, which tells her that this problem is a regression in product quality. She adds the following line to the Isolation section:

> *New to build 1.1.018; same test case passed against builds 1.1.007 through 1.1.017.*

Now that L. T. has a solid understanding of the bug, how it manifests itself, and how that affects the user, she writes the summary line for the bug report:

> *Summary*
>
> *Arial, Wingdings, and Symbol fonts corrupt new files.*

Here, L. T. has captured the impact to the user and the basic nature of the problem in a eight words. No one reading this summary will fail to understand what the problem is and why it matters.

At this point, the bug report is written. All that remains is to spend some time polishing it. First, L. T. condenses the report a bit by eliminating extra verbiage, for example rephrasing

> *If you save the file before Arializing the contents, the bug does not occur.*

as

> *Saving file before changing font prevents bug.*

and dropping the word "I" from the steps to reproduce.

Next, L. T. reviews the report for ambiguous or confusing wording. we'll see if any of the wording is ambiguous and might confuse the reader. She changes step three of from

> 3. Highlighted the text, then pulled down the font menu, and selected Arial.

to

> 3. Highlighted *all four lines of* text, then pulled down the font menu, and selected Arial.

Step four goes from

> 4. This nasty bug trashed all text into meaningless garbage, wasting the user's time. to

> 4. This nasty bug trashed all text into meaningless garbage, *including control characters, numbers, and other binary junk,* wasting the user's time. In the Isolation section, she

changes

> Also happens with Wingdings and Symbol fonts.

to

> *Reproduced with same steps using* Wingdings and Symbol fonts.

She adds the some detail to the Isolation line that reads

> This only happens under Windows 98.

expanding it to say that the bug

> Only happens under Windows 98*, not Solaris, Mac, or other Windows flavors.*

To wrap up, L. T. scours the report for any extreme statements or biased comments that might alarm or distract readers. She decides step four is a bit harsh, so she rephrases that as follows:

> 4. *All text converted to control characters, numbers, and other apparently random binary data.*

Finally, L.T. asks you to review the report. You like it, and ask her if perhaps this bug is related to the file-creation problem she found the first time around. You also ask her to remove the word "garbage" from the isolation section, being possibly inflammatory. The final bug report as submitted follows:

> Summary
>
> Arial, Wingdings, and Symbol fonts corrupt new files.
>
> Steps to Reproduce1. Started SpeedyWriter editor, then created new file.
>
> 2. Typed four lines of text, repeating "The quick fox jumps over the lazy brown dog" each time.
>
> 3. Highlighted all four lines of text, then pulled down the font menu, and selected Arial.
>
> 4. All text converted to control characters, numbers, and other apparently random binary data.
>
> 5. Reproduced three out of three tries.
>
> Isolation
>
> New to build 1.1.018; same test case passed against builds 1.1.007 through 1.1.017. *Possibly related to the file-creation bug fix in this release?*
>
> Reproduced with same steps using Wingdings and Symbol fonts.
>
> Saved file, closed SpeedyWriter and reopened file. *Still saw same corruption of the text.*

```
Saving file before changing font prevents bug.

Bug does not occur with existing files.

Only happens under Windows 98, not Solaris, Mac, or other
Windows flavors.
```

This is a good, solid failure description for the bug that L. T. has observed. It communicates the problem, it will help management decide whether to fix it based on impact to the customer, and it gives the development team some solid leads for their debugging activities.

Beyond the Failure Description

So far, I've focused on the failure description, which is the heart of the bug report. However, there are some other pieces of information commonly captured by bug tracking systems that are important to making the overall bug management process work. Two of these I call severity and priority. When I talk about "severity", I mean the technical impact on system under test. I often use the following scale for severity:

1. Data loss, hardware damage, or safety risk.
2. Loss of functionality without a reasonable workaround.
3. Loss of functionality with a reasonable workaround.
4. Partial loss of functionality or a feature set.
5. A cosmetic error.

Regarding priority, I mean the business importance, such as the impact on the project and the likely success of the product in the marketplace. I use the following scale for priority:

1. Must fix to proceed with the rest of the project effort, including testing.
2. Must fix for release; no customer will buy our product with this bug.
3. Fix desirable prior to release, as customers will object to the problem.
4. Time-to-market is definitely more important than fixing this bug, so fix it only if the release date not delayed.
5. Fix whenever convenient.

These are distinct concepts and priority is the more important one. However, all too often I have seen development teams get into heated arguments over whether a particular bug was "sev one" or "sev two", the subtext being that only severity one bugs matter. Such discussions are beside the point. When deciding whether to fix a bug, we should focus on how the associated failure modes affect the customers. Imagine a bug that involves display of an inappropriate message. That's a severity five bug on my scale. What if the inappropriate message includes a profane, lewd, or racist word? Less dramatically, suppose that the product name is displayed incorrectly in the splash screen? These kinds of problems are must-fix for release, severity notwithstanding. Conversely, a data loss bug that only occurs under certain obscure circumstances on unlikely and obsolete configurations may be severity one but priority five under my definition. What if our SpeedyWriter editor loses data when we save to a 360 kilobyte 5 ¼ inch floppy diskette? Perhaps, instead of fixing such a bug in the software, we should just document that our product doesn't support such drives.[5]

---

[5] Boris Beizer provides a good discussion of bugs and what he calls "bug importance" in *Software Testing Techniques*. Boris Beizer, Van Nostrand Reinhold, New York, 1990.

Another key piece of information for a bug report is the tester version of the software. In our case study, L. T. reported a bug against "`build 1.1.018`" and reported that "`builds 1.1.007 through 1.1.017`" did not suffer the same failure mode. I discussed the importance of test release management in a previous article.

In the case of integrate test management systems, where the bug and test repositories are in the same system, these two repositories can often be linked. Test case results can link to specific bug reports and bug reports can link to test cases. Such traceability is helpful for both testers and developers. If your bug tracking database doesn't support this, you may want to add to your process entering a test identifier into the failure description.

Speaking of traceability, some bugs can be tied to specific requirements, business rules, use cases, or quality risks. Again, you'll want to use whatever built-in support you have for such tracing, or, if your bug tracking system doesn't support it, add a step in your process where the tester notes this information in the failure description.

Since bugs can be quite dependent on the specific configuration tested, testers should make a habit of noting this as well. Ideally, your bug tracking system will support a look-up table for configurations, but, if not, you'll need a text description that captures the key items.

One last piece of information I have found useful to capture is affected subsystem, component, or function. For example, in the case of word processor we might have the following categories:

- User interface
- Edit engine
- Tools and utilities
- File operations
- Installation and configuration options
- Documentation and packaging
- Other
- Unknown
- N / A

Capturing this information is useful for various metrics, including tuning on-going and future testing to focus on those areas of the product that create the most problems.

In addition, bug tracking systems often capture the affected subsystem and the version of the system under test suffering from the problem. These fields allow you to classify the bug.

In a typical bug tracking system, some fields will be filled in by the developer who fixes the bug and possibly the release engineer who integrates the fixed code into the code repository. These fields don't typically have a lot of impact on the test team, except if they include a notation about which build contains the code fix associated with a given bug. Testers may find it useful to run reports on this information when preparing to confirmation test the bug fixes in a given test release.

Going beyond the solitary bug report, bug tracking information also has an aggregate dimension. Metrics related to find and fix rates, defect injection phases, closure period, root cause, subsystem affected, and the like are critical project dashboard data. A subsequent article will address issue of test project dashboards within the context of test status reporting, as this is another critical testing

process. In the meantime, for more information on bug tracking systems, what they should capture, and what metrics you can obtain from them, please see chapter four of my book.[6]

Bug Reporting Process Quality Indicators

So, based on what I've discussed so far, what are bug reporting process quality indicators?

Produces clear, concise bug reports that everyone understands

Since writing bug reports *is* writing—albeit very technical—the usual quality indicators for good technical writing apply. The writing should be clear and it should get to the point. It should be appropriate to the audience and speak to them on their terms. Where bug reports do contain jargon and acronyms, those should be well-understood project terms.

Documents bugs that get fixed

I opined earlier that bug reports are written to document problems, to help establish priority, and to provide information for fixing them. Kaner, Nguyen, and Falk put this slightly differently when they wrote that:

- The purpose of testing a program is to find problems in it.
- The purpose of finding problems is to get them fixed.[7]

While I think this definition may miss the broader risk-management function of testing group, I agree that the test team must show positive results of the testing investment in terms of increased product quality. The key metric here is the proportion of bugs reported by the test team that the development team fixes.

Of course, it takes two to tango, as the saying goes, and even the best bug reports won't get fixed if a recalcitrant development team disregards them and an incompetent management team ignores the ever-increasing bug backlog. (It happens.) However, disregarding those dysfunctional circumstances, when the bug reporting process provides managers with the information they need to prioritize bug fixing efforts and provides developers with the information they need to fix the bug reports assigned to them, that bug reporting process will have a better bug fix proportion than one which does not meet these criteria.

Has a low rate of duplication

Bug reports are about symptoms—observed anomalous behavior—while bug fixes address defects in systems. It is certainly the case that some disparate bug reports lead to the same underlying bug. However, in general this number should be fairly low. The situation to be avoided is that of two testers reporting the identical misbehavior in two separate bug reports. Such duplication of bug reports means duplication of effort by testers, developers, and managers.

As with the entire process, balance is required, though. I'd rather have two bug reports than no bug report, so I tell my testers not to spend more than five minutes searching the bug tracking system for an existing bug reports. In addition, I like to have testers circulate their bug reports via e-mail to the entire team, either as part of the review process or just for general information. I have found this level of effort keeps the duplication rate down to around five to ten percent, which I think is tolerable.

---

6 *Managing the Testing Process*. Rex Black, Microsoft Press, Seattle, 1999.

7 *Testing Computer Software*. Cem Kaner, Hung Nguyen, and Jack Falk. International Thompson Computer Press, Boston, 1993.

Minimizes bug report "ping-pong"

Bug report ping-pong is a game played between tester and developer where bug reports are kicked unproductively back and forth. Typical reasons for these ping-pong games are that the developer says a reported bug is irreproducible or is inappropriately assigned to her component or piece of code. The former problem can be resolved through careful bug reporting as described earlier, particular documenting the steps to reproduce and noting if the problem is intermittent. The latter problem requires dealing with the development manager to make sure that she—not you—is ultimately responsible for assigning bug reports to developers.

Helps delineate a clear boundary between testing and debugging

As Boris Beizer has pointed out, in the earliest stages of test effort maturity, testing is seen as an adjunct to the debugging process. As the testing effort matures, though, testing is seen as a distinct set of tasks devoted to locating bugs and managing quality risks.[8] In companies with independent test teams, testers focus on testing and developers handle the debugging tasks.

Let's take a quick look at the process associated with finding a failure, fixing the underlying bug, and confirming resolution as shown in Figure 1. The first three steps of this process are test activities, which belong to the test team, and occur as part of or in parallel with the bug reporting process. The next three steps are debugging activities, which belong to the development team, and happen as part of fixing the problem described in the bug report from the test team. The final step is confirmation and regression testing of the fix, which is again a testing activity. Therefore, this is a seven-step collaborative process that improves the quality of the system under test through closed-loop corrective action driven by bug reports. While the process is collaborative, each individual task belongs clearly to either the test team or the development team

Note that a good bug reporting process supports this separation of duties, because as I pointed out earlier, a good bug report gives the developer all the information he needs to reproduce the problem and start tracking down the bug. The bug reporting and bug management processes are the only vehicles required for communication between the two teams. The wall in the figure signifies this. I'm not suggesting that bug reports and bug fixes be "thrown over a wall" but rather that the process builds a wall that protects the developer from having to ask the tester questions about what her bug report means and the protects the tester from having to spend time duplicating effort reproducing the problem for the developer.

---

8 *Ibid.*

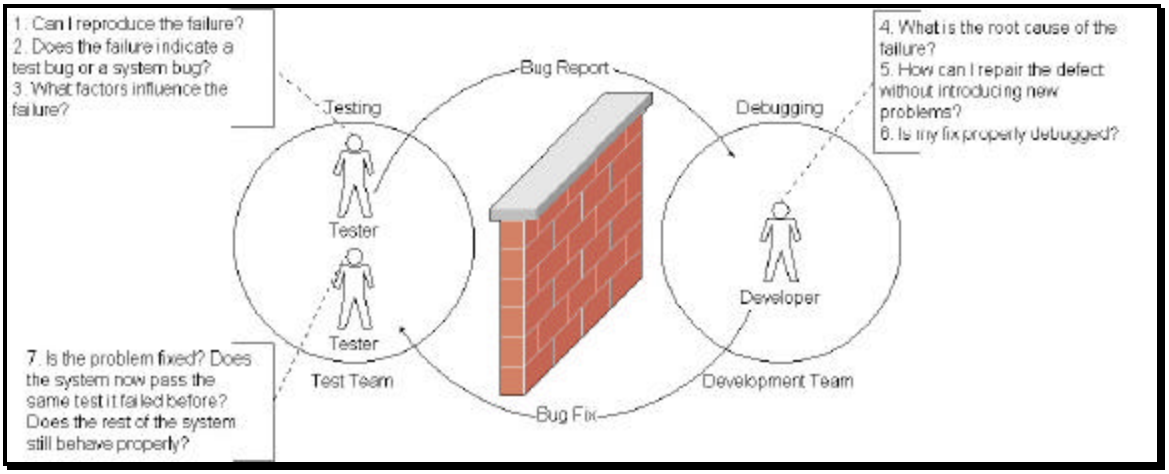

Figure 1: The find / debug / confirm process

Some test engineers consider me something of a killjoy for discouraging tester involvement in debugging, so let me explain why this is so important to me when I'm working as a test manager.

1. I have a lot of testing work for my testers to do. I'm held accountable for that work getting done. When testers help developers fix problems, they're not testing, which I define, during test execution periods, as running tests and reporting findings.
2. I can't manage my test operation when people loan themselves to the development organization at their discretion. Part of managing is allocating human resources to meet corporate objectives. I can't have re-allocation happening on the say-so of a tester who, frankly, may not have insight into the evolving priorities.
3. Debugging assistance by testers seldom a wise use of resources. Even if the tester is a competent programmer, presumably the reason he's testing, not programming, is that he's more valuable to the company in a testing role.
4. It boosts egos and feels like teamwork, but it doesn't build careers. I'm happy to structure someone's job to support a programming career path, usually by adding test toolsmith and test automation tasks to their workload. Such activities are much more conducive to learning about programming than participation in debugging, which is more programming tourism than a resume highlight.
5. As I hinted above, these activities often indicate a poor job of bug reporting being done by the tester. If the developer can't figure out what's going wrong based on the bug report and must ask the tester, then the solution is a crisper bug reporting process, not tester participation in debugging.

All that said, on some occasions testers should or even must participate in debugging, especially when unique test configurations or tools are required to reproduce the problem. It's really the exception to the rule, though, and a good bug reporting process minimizes the occurrence.

Doesn't use bug reports as a management problem escalation tool

Since we're writing bug reports to communicate about a problem in the system under test for the purposes of getting that problem prioritized and possibly fixed, we need to be sure that we use the communication channel the bug reporting and bug management processes create exclusively for that purpose. I avoid submitting bug reports that report failures of the *process* rather than the *product*. For example, if the release engineer didn't deliver the Monday morning build at 9:00 AM, I wouldn't write a bug report with the summary, "Build not delivered on time." I'd go talk to the release

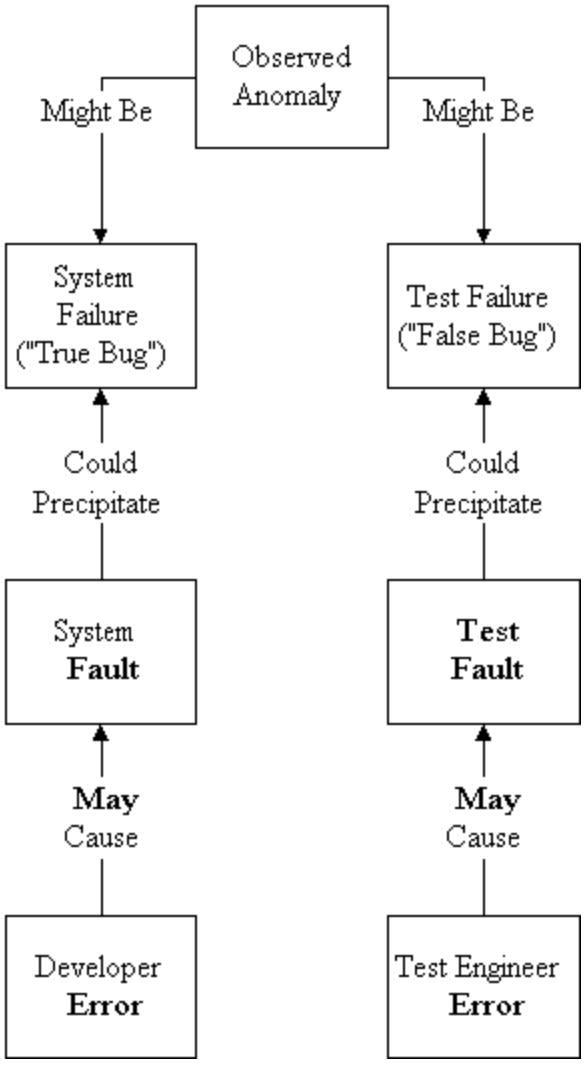

Figure 2: Confusing the source of the problem

engineer, explain the importance of timely receipt of builds, and, if the issue were an on-going situation, I'd also escalate to her manager. Of course, if the build was delivered on time but then wouldn't work, I might write a bug report titled, "Build 781 fails to install."

Assists distinguishing between test problems and SUT problems

As shown in Figure 2, an observed anomaly, rather than indicating a quality problem with the system under test, might actually be a testing problem. Such testing failures can arise with automated testing when the software driving the test fails, and it can arise for both automated and manual testing when the expected result is defined or predicted incorrectly (see below). There is no magic bullet for eliminating tester mistakes—if there were, it would work for developers, too, most likely, and we testers would be out of a job—but there are steps we can take. In terms of the bug reporting process, reproducing the problem and isolating the causes of the failure helps, as does the peer review at the end.

Supports the metrics effort (project dashboard)

As I mentioned in the last article, the test execution process needs to support gathering key metrics for managing testing and the project as a whole. Some of those metrics arise from the bug tracking system. So, the bug reports that your team writes must gather all the important information needed for these metrics. The process must work crisply, so that metrics that are driven by dates have accurate date information. Testers must take care to categories bugs appropriately—e.g., in terms of affected subsystem—so that metrics that analyze bugs by categories are meaningful. Above all, bug reports must be honest and complete. You're going to a great deal of trouble to report bugs, so you should make sure that the process supports gathering all the data you can as you do so.

The Broader Context of Bug Reporting

We don't run tests just to satisfy our intellectual curiosity about the quality of software. Likewise, even though bug reporting is an internal testing process, we don't write bug reports as a form of "software quality diary" for our own records. The bug reporting process—via the bug report management process—connects to the rest of the software development process. Earlier, I showed an example of this connection between the bug reporting process and the find/debug/confirm process.

For any given observed behavior, the tester must ask herself: Am I seeing the symptoms of a bug? If she answers that question "yes" and proceeds to write a bug report, three tangential questions arise. First, can we resolve ambiguity about correct behavior under particular conditions? In other words, how do we decide to report a bug when we have incomplete or conflicting information about how the system is supposed to behave? Because testers need to remain skeptical and play a "professional pessimist's" role in the development project, I espouse an active bias towards bug reporting. By this I mean that testers adopt the following attitudes and behaviors when executing a test:

- If in doubt, the testers assume the observed behavior is incorrect until they satisfy themselves otherwise;
- The testers report as a bug any situation where the on-screen help, user's guides, or any other documentation indicates that correct behavior differs from that observed;
- The testers **always** report any event leading to loss of data or the crash, wedge, hang, or other availability incident involving the software under test or the host system, no matter how intermittent or irreproducible;
- Finally, the testers report any circumstance where the system under test does not conform to reasonable expectations of program behavior or quality, or is otherwise confusing, misleading, or ambiguous.

This way of thinking must, however, be balanced against where you are in the project. In the early stages of testing, system quality is often very bad; testers should focus their efforts at this stage on writing the "big" bug reports, not worrying about misspellings in the user's guide, for example. Once the system improves, what is often criticized as nit-picky bug reporting is actually an appropriate attempt by testers to polish the fit and finish of the product. We should not be ashamed to do that at the appropriate time in the project.

The second question is whether we understand the effect, cause, and resolution of the observed aberrant behavior? In other words, do we know how the behavior in question will affect the customer's experience of quality, what the root cause of that behavior is, and where the problem can best be fixed? Let me give you an example: On one project, we found a bug that caused loss of data in files saved by applications on the system under test. The root cause was a hardware incompatibility related to the EIDE bus on the system under test and its hard drive. The problem was fixed by making a change in the computer's BIOS (basic input/output system, software that controls the operating system's interaction with the host hardware) that affected the timing of the OS's read and write operations to the hard drive. So, what was the bug? The lost data? The way the hard drive communicated with the system? (Or perhaps the way the system communicated with the hard drive?) The read and write code in the BIOS? From a user's perspective, perhaps we don't care—as long as the unacceptable behavior goes away—but from the point of view of the project these questions sometimes take on larger-than-life political dimensions.

The third question is whether we have an accurate "oracle"? That is, can the test system predict the correct outputs and behaviors for any given inputs and conditions presented to the system under test? The classic example of the oracle problem is the case of a testing a function that is to return the sine or logarithm of its argument. How do we verify correctness of the result without re-implementing in the test system the sine or logarithm functions? And, when we do implement an oracle for sines or logarithms, how do we know that when the oracle agrees with the system under test that the result is correct and, conversely, when the oracle disagrees with the system under test that the oracle is correct and the system under test is incorrect? In the former case, it is possible for a tester to report as passed a test that, in reality, failed. In the latter case, the test system might report a failure, and the tester might convince herself based on the test system's report that a bug exists in the system under test, where in reality the bug exists in the test system itself.

Kaner, Nguyen, and Falk do an excellent job of emphasizing the context in which we runs tests and write bug reports:

- The purpose of testing a program is to find problems in it.
- The purpose of finding problems is to get them fixed.[9]

The bug report is the vehicle by which these purposes are documented and then fulfilled. Figure XXX shows the three questions I posed above emanating from my starting question: Is the observed behavior a bug? Note that these questions move us out of the internal process of bug reporting into a collaborative process of measuring and improving product quality. The entire development team must work together and take the appropriate actions to answer all three related questions "yes" before the problem we have found, as represented by the bug report, can result in the desired improvement in product quality which occurs when the problem we have found is fixed.

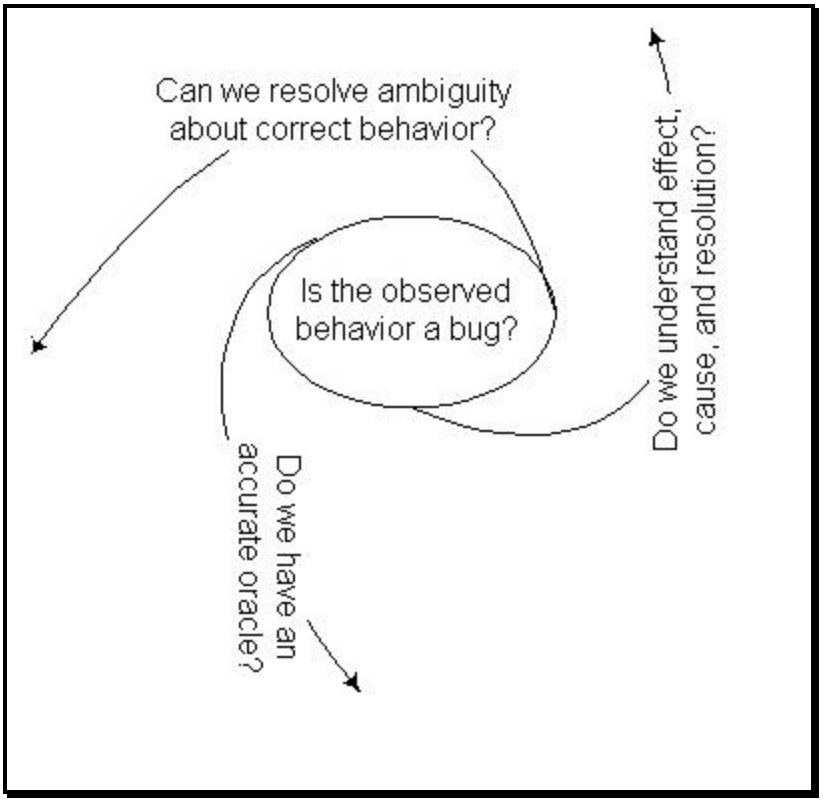

Figure 3: Tangents of the bug report

Handling Challenges

Writing good bug reports and managing those reports to closure presents the test team—and especially the test manager—with a number of challenges, even when you have the perfect process in place for writing these reports. These challenges are both technical (related to how hard it is intrinsically to report about the unknown) and political (related to the need to manage the interpersonal and team dynamics of criticizing the work of others).

---

9 *Testing Computer Software*. Cem Kaner, Hung Nguyen, and Jack Falk. International Thompson Computer Press, Boston, 1993.

Bug or feature?

As I mentioned earlier, ambiguity in—or a complete lack of—requirements or specifications can complicate the bug reporting process. When developers respond to bug reports with comments like, "Well, that's the way XYZ is supposed to work," testers often have no clear way to refute that claim. Rather than get ego-involved in the issue, my usual course of action is to try to resolve the ambiguity through discussions with the various concerned parties. People like the technical support manager, the sales and marketing team, the development manager, and others on the management team will probably have opinions as well. As long as you don't escalate every bug for such arbitration, you'll probably find these people ready, willing, and able to let you know how they think the product ought to behave.

Bugs that get fixed by accident

I once worked on a project where the development manager released new software daily. We chose not to accept builds into the test lab that often, as it disturbed our execution process to have to do confirmation testing that often. Every time we had a bug review meeting with this person, in response to any serious bug, whether his team had spent even a second trying to fix that bug or not, he would say, "Well, you need to retest that bug with release ABC because I bet we fixed that problem." Finally, I became so exasperated with this response that I told him, "You know, I'm sure you're convinced that you fixed all these bugs by accident, but in my experience a lot fewer bugs get fixed by accident than development managers would like. Until you actually expend some time trying to reproduce the problem and repair the bug, I am not going to ask my team to waste their time on retesting bugs."

I'll admit my delivery was a bit raw, but I stand by my comment in terms of course of action. Bugs don't typically get fixed by accident. In my experience, requests to retest every bug against every release in the off chance that somehow many of them magically went away are a form of development manager squid ink designed to obfuscate the fact that the product's quality is abysmally bad and/or to slow down the onslaught of bugs reported by the test team. Being a team player is good, but allowing yourself to be conned into squandering your test team's time retesting bugs that no one has spent a millisecond trying to fix is not exactly the pinnacle of good management.

Irreproducible bugs

As much difficult as this makes our lives as testers—and developers—some bugs just won't produce the same symptom again and again on command like a trained seal. For example, memory leaks depend on specific conditions arising repeatedly over an extended period of time before the system crashes or the application locks up. I once worked on a project where we had modem connections dropping sporadically, but couldn't make the problem occur deliberately. Ultimately, the developers had to put special probes in the software to track down the problem.

The important thing was that this level of effort was expended to find the bugs. Many times, intermittent bugs are also serious bugs. Crashes, losses of modem or LAN connections, sudden confusion on the system's part as to how to respond to an input can totally disrupt a user's work and even destroy her data.

So, be skeptical of claims that a bug "just went away" (see above) when neither developers nor testers can reproduce the bug. I usually keep such bugs open for a few releases (generally two or three weeks) to make sure we don't see it again. I ask developers what they want use to do if we do observe the problem. (For example, rebooting a hung system usually destroys the state data that might help a developer track down the bug; perhaps the developer will want to try to borrow into the system through the network to capture this information.) And make sure to keep a record of every time the bug *does* rear its ugly head again. Some sense of the kind of system unreliability that the bug

will cause is crucial to intelligent prioritization of the fix effort, and that means that you'll need to be able to report on its frequency.

Deferring trivia or creating test escapes?

Sometimes we report bugs that even we as testers consider nit-picky or insignificant. Perhaps we should just skip writing those reports? Or, when someone suggests that no one will case, we should just defer or cancel the report?

The problem with doing so is that we are arrogating ourselves the right to make decisions about what quality is for this product, but that decision really lies with the customer whose needs and requirements we as testers usually understand only imperfectly. We might decide to ignore a particular problem in the belief that no user would care, but then sales and marketing people can come back later and ask us how such an obvious bug escaped from the testing process. Once again, we must strive for a bug prioritization process that sees the impact of the problem through the customer's eyes, and make decisions about which bugs to fix and which to ignore based on that.

Build trust with developers

I once had a tester tell me that he enjoyed testing because he got a chance to "catch" developers. This kind of "gotcha" attitude can create real problems with bug reporting. If developers see testing as a game of tag—and having a bug report assigned against their components as the act of being tagged—then the "ping-pong" issue I discussed early is sure to arise.

In addition to keeping the team-oriented mindset during testing, four specific bug reporting actions can help create a confident attitude of cooperation between testers and developers.

1. Keep your cool during discussion about bug reports with developers.
2. Discuss bug reports with an open mind, accepting the possibility that you are mistaken, but also defend the reasonable expectations of customers.
3. Submit only quality bug reports and be open to suggestions for improving the quality of your bug reports.
4. Be flexible on delivery and reporting of bugs; e.g., if a developer wants a particular file or screen shot attached to a bug report—assuming the bug tracking system support such attachments—be cooperative.

There's really no overstating the kind of mischief that can occur when developers perceive the bug reporting process as a whip used by tester to beat them. In extreme cases, I have seen projects where developers and development managers demanded the right to review and approve bug reports before these reports were entered into the bug tracking system.

Let the development manager drive the bug fixing process

Sometimes test managers fall into the trap of haranguing individual developers to see when a particular bug will be fixed. This is usually a bad idea, for a couple reasons. First, since they don't report to you, you don't have a lot of insight into what other tasks they may have assigned. Interfering with their priorities won't endear you to them or to their manager. Second, you have little if any authority over them, one very likely outcome is that you will irritate people without achieving any useful result. You may be able to inveigle them to work on your favorite bug through bribery (lunch) or intellectual challenge (writing a particularly engrossing bug report), but taking the liberty of managing another's team generally won't work in the short- or long-term.

As a corollary to this observation, you should avoid using the bug report data to create charts or circulate reports that make individuals look bad. For example, many bug tracking tools have an "estimated fix date" field that developers fill in for their own manager to report when they expect a bug will be fixed. Running a report that shows, for each developer, which bugs they haven't yet fixed that are past due would be humiliating to people. Likewise, running a report component or affected subsystems by owner name—i.e., implying that the owner introduced the bug—would be both misleading and harmful to the targeted developers. As I mentioned in the previous subsection, developers must trust you with the bug tracking system, otherwise dysfunctional events occur.

Implementing Changes

This article has hopefully challenged some of your existing approaches for handling bugs, confirmed others, and maybe left you thinking of how you can implement some the ideas you like but haven't started yet. How you get your bug reporting process under control depends a lot on where you are now, but let me offer some suggestions.

1. I clearly have strong opinions about bug reporting, and you might, too. However, you and I have to be careful not to let our opinions blind us to the fact that other people are the customers for these bug reports. Talk to your customers—the technical support people, the developers, your fellow managers, and your test team—to find out what about the current process they like and dislike, as well as how they'd like to see the bug reports improved.
2. As I mentioned last time in the context of test execution, it's important to have a bug tracking system in place. I have a simple Microsoft Access database on the CD-ROM that comes with my book, *Managing the Testing Process*, which I'd be happy to e-mail to you if you don't have the time or resources to buy or build you own. You can use a spreadsheet, if absolutely necessary, but text files, word processing documents or collection of e-mails won't work. Remember, we want to support gathering metrics, and you won't be able to do any good analysis from these kinds of files.[10]
3. If you have an existing system in place, you should try to adapt it to support this process or some usable variant of it before deciding you must replace the system because you can't do everything you want. Changing your bug tracking system is a big decision, especially if you are in the middle of a project. The technical issues associated with moving data from one tool to another are not trivial—one of my associates and I once spent weeks doing just that—and there are political issues, too. Bug tracking systems are often shared with other groups, such as release management, development, and technical support, and managers of these groups often develop personal dashboards that use bug tracking data. At the right time, changing bug tracking systems can make sense, but I have been able to use this process even with what I consider one of the worst bug tracking tools on the market. Don't grab onto this tar baby unless you absolutely must, because you'll find, like Brer Rabbit, that freeing yourself from it is a lot harder than deciding you wanted to grab hold of it in the first place.
4. In some cases rigidity and consistency of process are critical, but in the case of bug reporting I have found that flexibility and sensitivity to priority are more important. On my teams, my only iron-clad rule on this process is that every bug report undergoes a peer review. In some cases, reproducing the problem three times makes sense, but in some cases it doesn't. For example, if you have found a very trivial problem and it takes an hour to reproduce it, is that a wise use of two hours?

---

[10] Okay, Unix buffs will probably say that you could do analysis on structured text files with filters like *grep* and *awk*, but, with simple relational database and spreadsheet applications so cheap, why would you spend your time this way?

5. Though people have commented to me that they don't see how to make the peer review work, I recommend that you give it a try before you dismiss it as a roadblock to efficient testing. Bad bug reports can waste a lot of time. Think of it this way: We testers are in the information business—we deliver an assessment of quality and risk to management and peers—so the last thing you'd want is for your one of your primary, highly-visible information products to be of inconsistent usefulness. Do you think that an editor reviews every word on the front page of the "Wall Street Journal" before it gets published? You bet! Peer reviews are proven methods for improving the quality of technical documents, and those of us in the quality business should use the tools at our disposal to do a good job.
6. The bug reporting process I've suggested is a lot of work. It might help to come up with a metric that can show the improvement of the process. For example, what is the reopen rate on bug reports? In other words, how many come back to test marked as fixed or irreproducible when they are not fixed properly or are easy to reproduce? Another metric might be the closure period—how long, on average, do bug reports stay open? A good bug reporting process should help drive both these numbers down.

Whatever changes you implement, patience and perseverance are important. People develop habits of working, especially when it comes to tasks they do over and over again. You'll need to keep this in mind as you work with these folks to improve your bug reporting processes.

Author Biography

Rex Black (Rex_Black@RexBlackConsulting.com) is the President and Principal Consultant of Rex Black Consulting Services, Inc. (http://www.RexBlackConsulting.com), an international software and hardware testing and quality assurance consultancy. He and his consulting associates help their clients with implementation, consulting, training, and staffing for testing and quality assurance projects. His book, *Managing the Testing Process*, was published in June, 1999, by Microsoft Press.