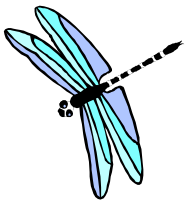
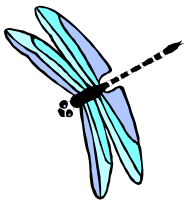


Bug Advocacy



How to

Win Friends,

influence programmers

and

STOMP BUGs.

(Not necessarily in that order.)

Cem Kaner
Professor of Computer Sciences
Florida Institute of Technology
kaner@kaner.com
www.kaner.com (testing website)
www.badsoftware.com (legal website)

Notice

These slides are modified from my seminar on software testing. That seminar is based on TESTING COMPUTER SOFTWARE (2nd Ed., a book co-authored with Jack Falk and Hung Quoc Nguyen), available from Wiley. The book presents additional material on bug analysis and the design of bug tracking processes. If you like the notes, you might try the book.

You can contact me by electronic mail at Cem Kaner <kaner@kaner.com>.

These notes include some legal information, but you are not my legal client and these notes do not provide specific legal advice. If you need legal advice, please consult your own attorney. I wrote these notes with the mass-market software development industry in mind. Mission-critical and life-critical software development efforts involve specific and rigorous procedures that are not described in these notes.

These are modified from the original course notes -- I've added a few pages of background to take into account some of the comments that I make in lecture.

Jack Falk and Hung Quoc Nguyen did much of the original work on this material. Hung has done extensive additional work on bug tracking system design. For details, go see www.logigear.com.

I thank James Bach, Elizabeth Hendrickson, Doug Hoffman, Bob Johnson, Brian Lawrence, Brian Marick, and Hung Quoc Nguyen for comments and contributions to earlier versions of these notes. However, any errors in these notes are mine.

GNU Free Documentation License Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

GNU Free Documentation License
Version 1.1, March 2000

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

GNU Free Documentation License Version 1.1, March 2000

How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document
> under the terms of the GNU Free Documentation License, Version 1.1
> or any later version published by the Free Software Foundation;
> with the Invariant Sections being LIST THEIR TITLES, with the
> Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.
> A copy of the license is included in the section entitled "GNU
> Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

FSF & GNU inquiries & questions to gnu@gnu.org.
Copyright notice above.
Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111, USA

Updated: 3 May 2000 mhatta

NOTE: The following was not included in this license (by Cem Kaner) even though it appears in the original GPL Free Documentation License, because I expect that all copies of these slides will be distributed in Opaque (PowerPoint, Star Office, or PDF) formats.

> **"If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public."**

About Cem Kaner

I'm in the business of improving software customer satisfaction.

I approach customer satisfaction from several angles. I've been a programmer, tester, writer, teacher, user interface designer, software salesperson, and a manager of user documentation, software testing, and software development, an organization development consultant and an attorney focusing on the law of software quality. These have provided many insights into relationships between computes, software, developers, and customers.

Current employment

- Professor of Software Engineering, Florida Institute of Technology
- Private practice in the Law Office of Cem Kaner

Books

- ***Testing Computer Software*** *(*1988; 2nd edition with Hung Nguyen and Jack Falk,1993). This received the *Award of Excellence* in the Society for Technical Communication's *Northern California Technical Publications Competition* and has the lifetime best sales of any book in the field.
- ***Bad Software: What To Do When Software Fails*** (with David Pels). Ralph Nader called this book "a how-to book for consumer protection in the Information Age."

Education

- J.D. (law degree, 1993). Elected to the American Law Institute, 1999.
- Ph.D. (experimental psychology, 1984) (trained in *measurement theory* and in *human factors*, the field concerned with making hardware and software easier and safer for humans to use).
- B.A. (primarily mathematics and philosophy, 1974).
- Certified in Quality Engineering (American Society for Quality, 1992). Examiner (1994, 1995) for the California Quality Awards.
- I am also a founder and co-host of the Los Altos Workshops on Software Testing and the Software Test Managers' Roundtable.

Bug Advocacy?

1. The point of testing is to find bugs.
2. **Bug reports are your primary work product.** This is what people outside of the testing group will most notice and most remember of your work.
3. The best tester isn't the one who finds the most bugs or who embarrasses the most programmers. The best tester is the one who gets the most bugs fixed.
4. Programmers operate under time constraints and competing priorities. For example, outside of the 8-hour workday, some programmers prefer sleeping and watching Star Wars to fixing bugs.

A bug report is a tool that you use to sell the programmer on the idea of spending her time and energy to fix a bug.

Note: When I say "the best tester is the one who gets the most bugs fixed," I am not encouraging bug counting metrics, which are almost always counterproductive. Instead, what I am suggesting is that the effective tester looks to the effect of the bug report, and tries to write it in a way that gives each bug its best chance of being fixed. Also, a bug report is successful if it enables an informed business decision. Sometimes, the best decision is to not fix the bug. The excellent bug report raises the issue and provides sufficient data for a good decision.

Selling Bugs

Time is in short supply. If you want to convince the programmer to spend her time fixing your bug, you may have to sell her on it.

> *(Your bug? How can it be your bug? The programmer made it, not you, right? It's the programmer's bug.* Well, yes, but you found it so now it's yours *too.)*

Sales revolves around two fundamental objectives:

- **Motivate the buyer (*Make her WANT to fix the bug.)***
- **Overcome objections (*Get past her excuses and reasons for not fixing the bug.)***

Motivating the Bug Fixer

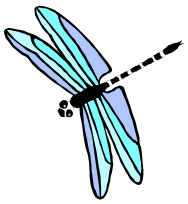
Some things that will often make programmers want to fix the bug:

- It looks really bad.
- It will affect lots of people.
- Getting to it is trivially easy.
- It has embarrassed the company, or a bug like it embarrassed a competitor.
- One of its cousins embarrassed the company or a competitor.
- Management (that is, someone with influence) has said that they really want it fixed.
- You've said that *you* want the bug fixed, and the programmer likes you, trusts your judgment, is susceptible to flattery from you, owes you a favor or accepted bribes from you.

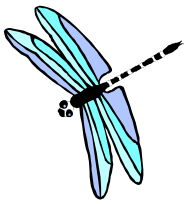
Overcoming Objections

Things that will make programmers resist spending their time on the bug:

- The programmer can't replicate the defect.
- Strange and complex set of steps required to induce the failure.
- Not enough information to know what steps are required, and it will take a lot of work to figure them out.
- The programmer doesn't understand the report.
- Unrealistic (e.g. "corner case")
- It will take a lot of work to fix the defect.
- A fix will introduce too much risk into the code.
- No perceived customer impact
- Unimportant (no one will care if this is wrong: minor error or unused feature.)
- Management doesn't care about bugs like this.
- The programmer doesn't like / trust you (or the customer who is complaining about the bug).



Bug Advocacy



Motivating Bug Fixes

By Better Researching

The Failure Conditions

Motivating The Bug Fix: Looking At The Failure

Some vocabulary

- An ***error*** (or ***fault)*** is a design flaw or a deviation from a desired or intended state.
- An error won't yield a failure without the ***conditions*** that trigger it. Example, if the program yields 2+2=5 on the 10th time you use it, you won't see the error before or after the 10th use.
- The ***failure*** is the program's actual incorrect or missing behavior under the error-triggering conditions.
- ***Defect*** *is frequently used to refer to the failure or to the underlying error.*

Nancy Leveson *(Safeware)* draws useful distinctions between errors, hazards, conditions, and failures.

Motivating The Bug Fix: Looking At The Failure

VOCABULARY EXAMPLE

Here's a defective program

```
INPUT A
INPUT B
PRINT A/B
```

What is the fault?

What is the critical condition?

What will we see as the failure?

Motivating the Bug Fix

When you run a test and find a failure, you're looking at a symptom, not at the underlying fault. You may or may not have found the best example of a failure that can be caused by the underlying fault.

Therefore you should do some follow-up work to try to prove that a defect:

- **is more serious than it first appears.**
- **is more general than it first appears.**

Motivating the Bug Fix: Make it More Serious

LOOK FOR FOLLOW-UP ERRORS

When you find a coding error, you have the program in a state that the programmer did not intend and probably did not expect. There might also be data with supposedly impossible values.

The program is now in a vulnerable state. Keep testing it and you might find that the *real* impact of the underlying fault is a much worse failure, such as a system crash or corrupted data.

I do three types of follow-up testing:

- **Vary my behavior (change the conditions by changing what I do)**
- **Vary the options and settings of the program (change the conditions by changing something about the program under test).**
- **Vary the software and hardware environment.**

Motivating the Bug Fix: Make it More Serious

Follow-Up: Vary Your Behavior

Keep using the program after you see the problem.

- Bring it to the failure case again (and again). If the program fails when you do X, then do X many times. Is there a cumulative impact?
- Try things that are related to the task that failed. For example, if the program unexpectedly but slightly scrolls the display when you add two numbers, try tests that affect adding or that affect the numbers. Do X, see the scroll. Do Y then do X, see the scroll. Do Z, then do X, see the scroll, etc. (If the scrolling gets worse or better in one of these tests, follow that up, you're getting useful information for debugging.)
- Try things that are related to the failure. If the failure is unexpected scrolling after adding, try scrolling first, then adding. Try repainting the screen, then adding. Try resizing the display of the numbers, then adding.
- Try entering the numbers more quickly or changing the speed of your activity in some other way.
- And try the usual exploratory testing techniques. So, for example, you might try some interference tests. Stop the program or pause it or swap it just as the program is failing. Or try it while the program is doing a background save. Does that cause data loss corruption along with this failure?

Motivating the Bug Fix: Make it More Serious

Follow-Up: Vary Options and Settings

In this case, the steps to achieve the failure are taken as given. Try to reproduce the bug when the program is in a different state:

- Use a different database.
- Change the values of persistent variables.
- Change how the program uses memory.
- Change anything that looks like it might be relevant that allows you to change as an option.

For example, suppose the program scrolls unexpectedly when you add two numbers. Maybe you can change the size of the program window, or the precision (or displayed number of digits) of the numbers, or background the activity of the spell checker.

Motivating the Bug Fix: Make it More Serious

Follow-Up: Vary Options and Settings

A bug might show a more serious failure if you run the program with less memory, a higher resolution printer, with more (or fewer) device interrupts coming in etc.

- *If there is anything involving timing, use a really slow computer, a really slow link, a really slow modem or printer. And use very fast ones.*
- *If there is a video problem, try higher resolutions on the video card. Try displaying MUCH more complex images (and much simpler ones).*

Note that we are not:

- checking standard configurations
- asking how broad the range of circumstances is that produces the bug.

What we're asking is whether there is a particular configuration that will show the bug more spectacularly.

Returning to the example (unexpected scrolling when you add two numbers), try things like:

- Different video resolutions
- Different mouse settings if you have a wheel mouse that does semi-automated scrolling
- An NTSC (television) signal output instead of a traditional (XGA or SVGA, etc.) monitor output.

Motivating the Bug Fix: Make it More Serious

IS THIS BUG NEW TO THIS VERSION?

In many projects, an old bug (from a previous shipping release of the program) might not be taken very seriously if there weren't lots of customer complaints.

- (If you know it's an old bug, check its complaint history.)
- The bug will be taken more seriously if it is new.
- You can argue that it should be treated as new if you can find a new variation or a new symptom that didn't exist in the previous release. What you are showing is that the new version's code interacts with this error in new ways. That's a new problem.
- This type of follow-up testing is especially important during a maintenance release that is just getting rid of a few bugs. Bugs won't be fixed unless they were (a) scheduled to be fixed because they are critical or (b) new side effects of the new bugfixing code.

Motivating the Bug Fix: Show it is More General

Question: How many programmers does it take to change a light bulb?

Answer: *What's the problem? The bulb at my desk works fine!*

LOOK FOR CONFIGURATION DEPENDENCE

Bugs that don't fail on the programmer's machine are much less credible (to that programmer). If they are configuration dependent, the report will be much more credible if it identifies the configuration dependence directly (and so the programmer starts out with the expectation that it won't fail on all machines.)

In the ideal case (standard in many companies), you test on 2 machines

- Do your main testing on Machine 1. Maybe this is your powerhouse: latest processor, newest updates to the operating system, fancy printer, video card, USB devices, huge hard disk, lots of RAM, cable modem, etc.
- When you find a defect, use Machine 1 as your bug reporting machine and replicate on Machine 2. Machine 2 is totally different. Different processor, different keyboard and keyboard driver, different video, barely enough RAM, slow, small hard drive, dial-up connection with a link that makes turtles look fast.
- Some people do their main testing on the turtle and use the power machine for replication.
- Write the steps, one by one, on the bug form at Machine 1. As you write them, try them on Machine 2. If you get the same failure, you've checked your bug report while you wrote it. (A valuable thing to do.)
- If you don't get the same failure, you have a configuration dependent bug. Time to do troubleshooting. But at least you know that you have to.

AS A MATTER OF GENERAL GOOD PRACTICE, IT PAYS TO REPLICATE EVERY BUG ON A SECOND MACHINE.

Motivating the Bug Fix: Show it is More General

TRY VARIANTS THAT SHOULDN'T MATTER

The point of this exercise is to show that you get the same failure whether a given variable is set one way or another.

In follow-up testing, we varied "irrelevant" variables with an eye to seeing differences in the failure symptoms. We picked variables that looked promising for this.

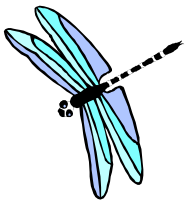
In generalization testing, we're still looking to see whether the failure symptoms change, but we're picking variables that we don't expect to cause a change. The point is to take a variable that has been set one way throughout the testing of this bug, show that you get the same problem with a different setting, and then you can ignore this variable, not discuss it in the bug report, treat it as irrelevant.

Motivating the Bug Fix: Show it is More General

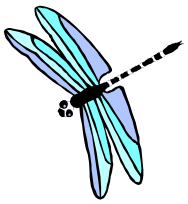
UNCORNER YOUR CORNER CASES

We test at extreme values because these are the most likely places to show a defect. ***But once we find the defect, we don't have to stick with extreme value tests.***

- Try mainstream values. These are easy settings that should pose no problem to the program. Do you replicate the bug? If yes, write it up, referring primarily to these mainstream settings. This will be a very credible bug report.
- If the mainstream values don't yield failure, but the extremes do, then do some troubleshooting around the extremes. Is the bug tied to a single setting (a true corner case)? Or is there a small range of cases? What? In your report, identify the narrow range that yields failures. The range might be so narrow that the bug gets deferred. That might be the right decision. In some companies, the product has several hundred open bugs a few weeks before shipping. They have to decide which 300 to fix (the rest will be deferred). Your reports help the company choose the right 300 bugs to fix, and help people size the risks associated with the remaining ones.



Bug Advocacy



Overcoming

OBJECTIONS

By Better Researching The Failure Conditions

Overcoming Objections Via Analysis of the Failure

Things that will make programmers resist spending their time on the bug:

- **The programmer can't replicate the defect.**
  - Strange and complex set of steps required to induce the failure.
  - Not enough information to know what steps are required, and it will take a lot of work to figure them out.
  - The programmer doesn't understand the report.
  - Unrealistic (e.g. "corner case")

Objection, Objection: Non-Reproducible Errors

Always report non-reproducible errors. If you report them well, programmers can often figure out the underlying problem.

To help them, you must describe the failure as precisely as possible. If you can identify a display or a message well enough, the programmer can often identify a specific point in the code that the failure had to pass through.

- When you realize that you can't reproduce the bug, write down everything you can remember. Do it now, before you forget even more. As you write, ask yourself whether you're sure that you did this step (or saw this thing) exactly as you are describing it. If not, say so. Draw these distinctions right away. The longer you wait, the more you'll forget.
- Maybe the failure was a delayed reaction to something you did before starting this test or series of tests. Before you forget, note the tasks you did before running this test.
- Check the bug tracking system. Are there similar failures? Maybe you can find a pattern.
- Find ways to affect timing of your program or of your devices, Slow down, speed up.
- ***Talk to the programmer*** and/or read the code.

Non-Reproducible Errors

- The fact that a bug is not reproducible is data. The program is telling you that you have a hole in your logic. You are not entertaining certain relevant conditions. Why not?
- See Watts Humphrey, *Personal Software Process*, for recommendations to programmers of a system for discovering and then eliminating characteristic errors from their code. A non-reproducible bug is a tester's error, just like a design bug is a programmer's error. It's valuable to develop a system for discovering your blind spots. To improve over time, keep track of the bugs you're missing and what conditions you are not attending to (or find too hard to manipulate).
- The following pages give a list of some conditions commonly ignored or missed by testers. Your personal list will be different in some ways, but maybe this is a good start. When you run into a irreproducible defect look at this list and ask whether any of these conditions could be the critical one. If it could, vary your tests on that basis and you might reproduce the failure.

(Note: Watts Humphrey suggested to me the idea of keeping a list of commonly missed conditions. It has been a tremendously valuable insight.)

Non-Reproducible Errors: Examples of Conditions Often Missed

- Some problems have delayed effects:
  - » a memory leak might not show up until after you cut and paste 20 times.
  - » stack corruption might not turn into a stack overflow until you do the same task many times.
  - » a wild pointer might not have an easily observable effect until hours after it was mis-set.

If you suspect that you have time-delayed failures, use tools such as videotape, capture programs, debuggers, debug-loggers, or memory meters to record a long series of events over time.

(I highlight these three in lecture because so many people have trouble with time-delayed bugs.Until you think backwards in time and ask how you could find a defect that has a delayed reaction effect, you won't be able to easily recreate these problems.

The following pages give additional examples. There are plenty of other conditions that are relevant in your environment. Start with these but add others as you learn of them. How do you learn? Sometimes, someone will fix a bug that you reported as non-reproducible. Call the programmer, ask him how to reproduce it, what are the critical steps that you have to take? You need to know this anyway, so that you can confirm that a bug fix actually worked.)

Non-Reproducible Errors: Examples of Conditions Often Missed

- The bug depends on the value of a *hidden input variable*. (Bob Stahl teaches this well.) In any test, there are the variables that we think are relevant and then there is everything else. If the data that you think are relevant don't help you reproduce the bug, ask what other variables were set, and what their values were, in the course of running or preparing this test.
- Some conditions are hidden and others are invisible. You cannot manipulate them and so it is harder to recognize that they're present. You might have to talk with the programmer about what state variables or flags get set in the course of using a particular feature.
- Some conditions are *catalysts*. They make failures more likely to be seen. Example: low memory for a leak; slow machine for a race. But sometimes catalysts are more subtle, such as use of one feature that has a subtle interaction with another.
- Some bugs are predicated on corrupted data. They don't appear unless there are impossible configuration settings in the config files or impossible values in the database. What could you have done earlier today to corrupt this data?
- The bug might appear only at a specific time of day or day of the month or year. Look for week-end, month-end, quarter-end and year-end bugs, for example.
- Programs have various degrees of *data coupling*. When two modules use the same variable, oddness can happen in the second module after the variable is changed by the first. (Books on structured design, such as Yourdon/Constantine often analyze different types of coupling in programs and discuss strengths and vulnerabilities that these can create.) In some programs, interrupts share data with main routines in ways that cause bugs that will only show up after a specific interrupt.
- Special cases appear in the code because of time or space optimizations or because the underlying algorithm for a function depends on the specific values fed to the function (talk to your programmer).
- The bug depends on you doing related tasks in a specific order.
- The bug is caused by a race condition or other time-dependent event, such as:
  - » An interrupt was received at an unexpected time.
  - » The program received a message from another device or system at an inappropriate time (e.g. after a time-out.)
  - » Data was received or changed at an unexpected time.
- The bug is caused by an error in error-handling. You have to generate a previous error message or bug to set up the program for this one.

Non-Reproducible Errors: Why is this Bug Hard to Reproduce?

- *Time-outs* trigger a special class of multiprocessing error handling failures. These used to be mainly of interest to real-time applications, but they come up in client/server work and are very pesky.

  Process A sends a message to Process B and expects a response. B fails to respond. What should A do? What if B responds later?
- Another inter-process error handling failure -- Process A sends a message to B and expects a response. B sends a response to a different message, or a new message of its own. What does A do?
- You're being careful in your attempt to reproduce the bug, and you're typing too slowly to recreate it.
- The program might be showing an initial state bug, such as:
  - » The bug appears only the first time after you install the program (so it happens once on every machine.)
  - » The bug appears once after you load the program but won't appear again until you exit and reload the program.

    (See Testing Computer Software's Appendix's discussion of initial state bugs.)
- The program may depend on one version of a DLL. A different program loads a different version of the same DLL into memory. Depending on which program is run first, the bug appears or doesn't.
- The problem depends on a file that you think you've thrown away, but it's actually still in the Trash (where the system can still find it).
- A program was incompletely deleted, or one of the current program's files was accidentally deleted when that other program was deleted. (Now that you've reloaded the program, the problem is gone.)
- The program was installed by being copied from a network drive, and the drive settings were inappropriate or some files were missing. (This is an invalid installation, but it happens on many customer sites.)
- The bug depends on co-resident software, such as a virus checker or some other process, running in the background. Some programs run in the background to intercept foreground programs' failures. These may sometimes trigger failures (make errors appear more quickly).

Non-Reproducible Errors: Why is this Bug Hard to Reproduce?

- You forgot some of the details of the test you ran, including the critical one(s) or you ran an automated test that lets you see that a crash occurred but doesn't tell you what happened.
- The bug depends on a crash or exit of an associated process.
- The program might appear only under a peak load, and be hard to reproduce because you can't bring the heavily loaded machine under debug control (perhaps it's a customer's system).
- On a multi-tasking or multi-user system, look for spikes in background activity.
- The bug occurred because a device that it was attempting to write to or read from was busy or unavailable.
- It might be caused by keyboard keybounce or by other hardware noise.
- Code written for a cooperative multitasking system can be thoroughly confused, sometimes, when running on a preemptive multitasking system. (In the *cooperative* case, the foreground task surrenders control when it is ready. In the *preemptive* case, the operating system allocates time slices to processes. Control switches automatically when the foreground task has used up its time. The application is suspended until its next time slice. This switch occurs at an arbitrary point in the application's code, and that can cause failures.
- The bug occurs only the first time you run the program or the first time you do a task after booting the program. To recreate the bug, you might have to reinstall the program. If the program doesn't uninstall cleanly, you might have to install on a fresh machine (or restore a copy of your system taken before you installed this software) before you can see the problem.
- The bug is specific to your machine's hardware and system software configuration. (This common problem is hard to track down later, after you've changed something on your machine. That's why good reporting practice involves replicating the bug on a second configuration.)
- The bug was a side-effect of a hardware failure. This is rarely the problem, but sometimes it is. A flaky power supply creates irreproducible failures, for example. Another example: One prototype system had a high rate of irreproducible firmware failures. Eventually, these were traced to a problem in the building's air conditioning. The test lab wasn't being cooled, no fan was blowing on the unit under test, and several of the prototype boards in the machine ran very hot. (Later versions would run cooler, but these were early prototypes.) The machine was failing at high temperatures.
- Someone tinkered with your machine when you weren't looking.

Putting Bugs in the Dumpster

Problem:

Non-reproducible bugs burn a huge amount of programmer troubleshooting time, then get closed. Until they are closed, they show up in open-bug statistics. In companies that manage more by bug numbers than by good sense, there is tremendous pressure to close irreproducible bugs quickly.

The Dumpster:

- A resolution code that puts the bug into an ignored storage place. The bug shows up as resolved (or is just never counted) in the bug statistics, but it is not closed. It is in a holding pattern.
- Assign a non-reproducible bug to the dumpster whenever you (testers and programmers) spend enough time on it that you don't think that more work on the bug will be fruitful *at this time.*

Dumpster Diving

- Every week or two, (testers and/or programmers) go through the dumpster bugs looking for similar failures. At some point, you'll find a collection of several similar reports. If you (or the programmer) think there are enough variations in the reports to provide useful hints on how to repro the bug, spend time on the collection. If you (or the programmer) can repro the bugs, reopen them with the extra info (status is now open, resolution is pending)
- Near the end of the project, do a final review of bugs in the dumpster. These will either close non-repro or be put through one last scrutiny

Overcoming Objections Via Analysis of the Failure

Things that will make programmers resist spending their time on the bug:

- The programmer can't replicate the defect.
- Strange and complex set of steps required to induce the failure.
- Not enough information to know what steps are required, and it will take a lot of work to figure them out.
- The programmer doesn't understand the report.
- Unrealistic (e.g. "corner case")

Simplify, Simplify: Split the Report in Two

If you see two failures, write two reports.

Combining failures on one report creates problems:

- The summary description is typically vague. You say words like “fails” or “doesn’t work” instead of describing the failure more vividly. This weakens the impact of the summary.
- The detailed report is typically lengthened. It’s common to see bug reports that read like something written by an inept lawyer. Do this unless that happens in which case don’t do this unless the first thing and then the testcase of the second part and sometimes you see this but if not then that.
- Even if the detailed report is rationally organized, it is longer (there are two failures and two sets of conditions, even if they are related) and therefore more intimidating.
- You’ll often see one bug get fixed but not the other.
- When you report related problems on separate reports, it is a courtesy to cross-reference them.

Simplify, Simplify: Eliminate Unnecessary Steps

Sometimes it's not immediately obvious what steps can be dropped from a long sequence of steps in a bug.

> *Look for critical steps -- Sometimes the first symptoms of an error are subtle.*

You have a list of all the steps that you took to show the error. You're now trying to shorten the list. Look carefully for any hint of an error as you take each step -- A few things to look for:

- Error messages (you got a message 10 minutes ago. The program didn't fully recover from the error, and the problem you see now is caused by that poor recovery.)
- Delays or unexpectedly fast responses.
- Display oddities, such as a flash, a repainted screen, a cursor that jumps back and forth, multiple cursors, misaligned text, slightly distorted graphics, doubled characters, omitted characters, or display droppings (pixels that are still colored even though the character or graphic that contained them was erased or moved).
- Sometimes the first indicator that the system is working differently is that it sounds a little different than normal.
- An in-use light or other indicator that a device is in use when nothing is being sent to it (or a light that is off when it shouldn't be).
- Debug messages—turn on the debug monitor on your system (if you have one) and see if/when a message is sent to it.

If you've found what looks like a critical step, try to eliminate almost everything else from the bug report. Go directly from that step to the last one (or few) that shows the bug. If this doesn't work, try taking out individual steps or small groups of steps.

Simplify, Simplify: Put Variations After the Main Report

Suppose that the failure looks different under slightly different circumstances. For example:

- The timing changes if you do a additional two sub-tasks before hitting the final reproduction step
- The failure won't show up at all or is much less serious if you put something else at a specific place on the screen
- The printer prints different garbage (instead of the garbage you describe) if you make the file a few bytes longer

This is all useful information for the programmer and you should include it. But to make the report clear:

- Start the report with a simple, step-by-step description of the shortest series of steps that you need to produce the failure.
- Identify the failure. (Say whatever you have to say about it, such as what it looks like or what impact it will have.)
- ***Then*** **add a section that says "ADDITIONAL CONDITIONS"** and describe, one by one, in this section the additional variations and the effect on the observed failure.

Corner Conditions
A Thought Experiment

(I've never tried anything like this on Quicken. This example is purely hypothetical.)

- Imagine that you were testing Quicken. I understand that it can handle an arbitrarily large number of checks, limited only by the size of the hard disk. So, being a tester, you go buy the biggest whopping hard disk on the market (about 70 gig at the moment, I think) and then write a little program to generate a Quicken-like data file with a billion or so checks.
- Now, open Quicken, load that file, and add 1 more check. Let's pretend that it works — sort of. The clock spins and spins while Q sorts the new check into place. At the end of four days, you regain control.
- So you write a bug report— "4 days to add a check is a bit too long."
- The programmers respond— "Not a bug. Only commercial banks manage a billion checks. If you're Citibank, use a different program."

Suppose that you decided to follow this up because you thought the programmers are dismissing a real problem without analysis. What would you do?

Overcoming Objections: Unrealistic (e.g., Corner Conditions)

Some reports are inevitably dismissed as unrealistic (having no importance in real use).

- If you're dealing with an extreme value, do follow-up testing with less extreme values.
- If you're protesting a bug that has been left unfixed for several versions, realized that it has earned tenure in some people's minds. Perhaps, though, customer complaints about this bug have simply never filtered through to developers.
- If your report of some other type of defect or design issue is dismissed as having "no customer impact," ask yourself:

  Hey, how do they know the customer impact?

- Then check with
  - » Technical marketing
  - » Technical support
  - » Human factors
  - » Documentation and training
  - » Network administrators
  - » In-house power users
  - » Maybe sales or corporate communications.

Thought Experiment--Notes

The classroom discussion on the thought experiment walks through a lot of alternatives. Here is the result that I try to steer people toward:

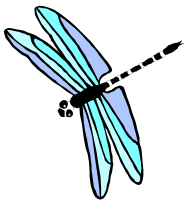
(a) There is no wisdom in following up a deferred bug unless you can add value to your report. The operative rule of thumb here is, "**If you're going to fight, win!**" (If you can't win, don't fight. Pick your battles.)

(b) There isn't much sense in doing a full engineering study at this point (some people want to do a parametric map showing how much time it takes to add a cheque, across many different numbers of cheques. The reason this is not sensible is that the bug has already been deferred. It will probably stay deferred. The more time you spend on it, the more time you put at risk. So, the question is, **how do we spend the minimum time needed to mount a good challenge?**

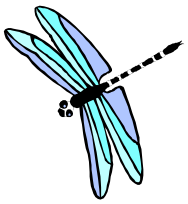
(c) We are looking for a single, magic number of cheques, if possible. That magic number is the largest credible number of cheques, i.e. a number small enough that the programmers won't dismiss it but large enough that the program still has a good chance of showing a problem.

(d) Many testers draw on their own experience for deciding what the right number of cheques would be. There are worse things to do, but this is certainly subject to the challenge that the tester is using a dumb number.

(e) In some companies, the appropriate person to contact is the marketing manager. But not always. **The key point is to look for *the credible source*. The "credible source" is the person who can say that X is the requirement and people will believe her**. Sometimes this is the marketing manager, sometimes the support manager, sometimes the founder of the company, sometimes a specific tester.

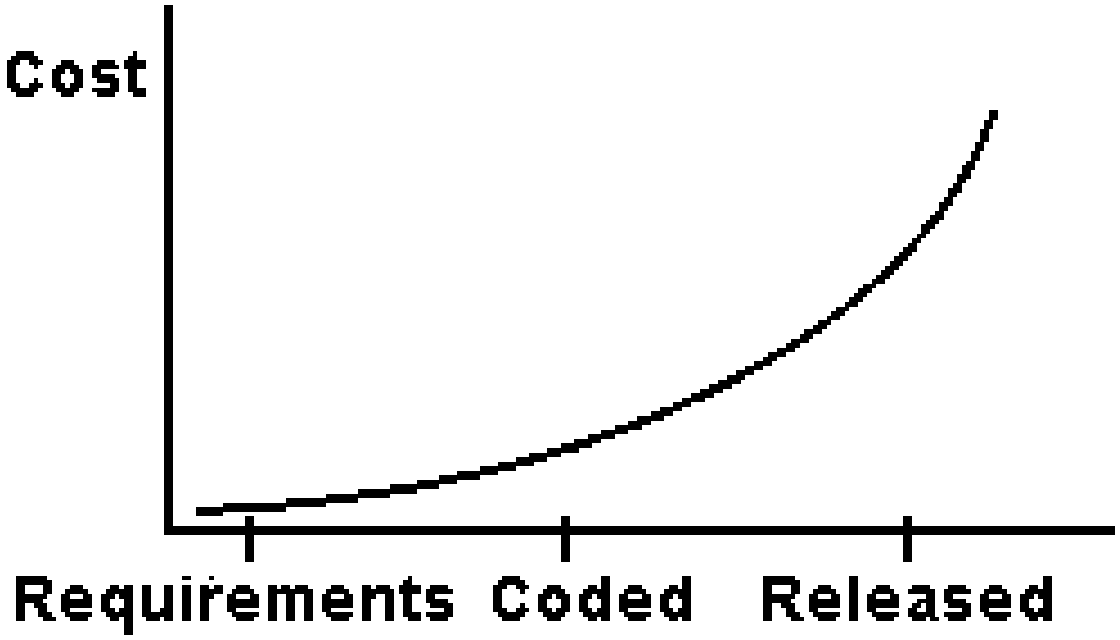


Bug Advocacy



Advocating for bug fixes by alerting people to costs.

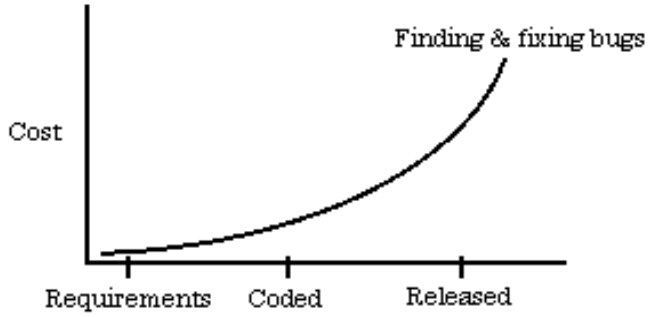
Money Talks: Cost of Finding and Fixing Software Errors



This curve maps the increase of cost the later that you find and fix an error.

This is the most commonly taught cost curve in software engineering. Usually people talk about it from the developers-eye view. That is, the discussion centers around how much it costs to find the bug, how much it costs to fix the bug, and how much it costs to distribute the bug fix. But sometimes, it pays to adopt the viewpoints of other stakeholders, who might stand to lose more money than the development and support organizations.

Cost of Finding and Fixing Software Errors



These costs escalate because more people in and out of the company are affected by bugs, and more severely affected, as the product gets closer to release. We all know the obvious stuff

- if we find bugs in requirements, we can fix them without having to recode anything;
- programmers who find their own bugs can fix them without taking time to file bug reports or explain them to someone else;
- it is hugely expensive to deal with bugs in the field (in customers' hands).

Along with this, there are many effects on other stakeholders in the company. For example, think of the marketing assistant who wastes days trying to create a demo, but can't because of bugs.

Quality Cost Analysis

Quality Cost Measurement is a cost control system used to identify opportunities for reducing the controllable quality-related costs

The *Cost of Quality* is the total amount the company spends to achieve and cope with the quality of its product.

This includes the company's investments in improving quality, and its expenses arising from inadequate quality.

A key goal of the quality engineer is to help the company minimize its cost of quality.

» Refer to the paper, "Quality Cost Analysis: Benefits & Risks."

Quality-Related Costs

Prevention	***Appraisal***
Cost of preventing customer dissatisfaction, including errors or weaknesses in software, design, documentation, and support.	Cost of inspection (testing, reviews, etc.).
Internal failure	***External failure***
Cost of dealing with errors discovered during development and testing. Note that the company loses money as a user (who can't make the product work) and as a developer (who has to investigate, and possibly fix and retest it).	Cost of dealing with errors that affect your customers, after the product is released.

Examples of Quality Costs

Prevention	***Appraisal***
• Staff training • Requirements analysis • Early prototyping • Fault-tolerant design • Defensive programming • Usability analysis • Clear specification • Accurate internal documentation • Pre-purchase evaluation of the reliability of development tools	• Design review • Code inspection • Glass box testing • Black box testing • Training testers • Beta testing • Test automation • Usability testing • Pre-release out-of-box testing by customer service staff
Internal Failure	***External Failure***
• Bug fixes • Regression testing • Wasted in-house user time • Wasted tester time • Wasted writer time • Wasted marketer time • Wasted advertisements • Direct cost of late shipment • Opportunity cost of late shipment	• Technical support calls • Answer books (for Support) • Investigating complaints • Refunds and recalls • Interim bug fix releases • Shipping updated product • Supporting multiple versions in the field • PR to soften bad reviews • Lost sales • Lost customer goodwill • Reseller discounts to keep them selling the product • Warranty, liability costs

Customers' Quality Costs

Seller: external costs	*Customer: failure costs*
These are the types of costs absorbed by the seller that releases a defective product.	***These are the types of costs absorbed by the customer who buys a defective product.***
• Technical support calls • Preparing answer books • Investigating complaints • Refunds and recalls • Interim bug fix releases • Shipping updated product • Supporting multiple versions in the field • PR to soften harsh reviews • Lost sales • Lost customer goodwill • Reseller discounts to keep them selling the product • Warranty, liability costs • Gov't investigations	• Wasted time • Lost data • Lost business • Embarrassment • Frustrated employees quit • Demos or presentations to potential customers fail because of the software • Failure during tasks that can only be done once • Cost of replacing product • Reconfiguring the system • Cost of recovery software • Cost of tech support • Injury / death

One way to make an argument based on customer costs is to evaluate costs to an in-house group of users.

Influencing Others Based on Costs

It's probably impossible to fix every bug. Sometimes the development team will choose to not fix a bug based on their assessment of its risks for them, without thinking of the costs to other stakeholders in the company.

- Probable tech support cost.
- Risk to the customer.
- Risk to the customer's data or equipment.
- Visibility in an area of interest to reviewers.
- Extent to which the bug detracts from the use of the program.
- How often will a customer see it?
- How many customers will see it?
- Does it block any testing tasks?
- Degree to which it will block OEM deals or other sales.

To argue against a deferral, ask yourself which stakeholder(s) will pay the cost of keeping this bug. Flag the bug to them.

Cem Kaner, Ph.D., J.D.
Attorney at Law
kaner@kaner.com

P.O. Box 1200
Santa Clara, CA 95052
408-244-7000

Quality Cost Analysis: Benefits and Risks

Published in *Software QA*, , 3, #1, 1996, p. 23.

> "Because the main language of [corporate management] was money, there emerged the concept of studying quality-related costs as a means of communication between the quality staff departments and the company managers."[1]

Joseph Juran, one of the world's leading quality theorists, has been advocating the analysis of quality-related costs since 1951, when he published the first edition of his *Quality Control Handbook*. Feigenbaum made it one of the core ideas underlying the Total Quality Management movement.[2] It is a tremendously powerful tool for product quality, including software quality.

What is Quality Cost Analysis?

Quality costs are the costs associated with preventing, finding, and correcting defective work. These costs are huge, running at 20% - 40% of sales.[3] Many of these costs can be significantly reduced or completely avoided. One of the key functions of a Quality Engineer is the reduction of the total cost of quality associated with a product.

Here are six useful definitions, as applied to software products. Figure 1 gives examples of the types of cost. Most of Figure 1's examples are (hopefully) self-explanatory, but I'll provide some additional notes on a few of the costs:[4]

- ***Prevention Costs:*** Costs of activities that are specifically designed to prevent poor quality. Examples of "poor quality" include coding errors, design errors, mistakes in the user manuals, as well as badly documented or unmaintainably complex code.

  Note that most of the prevention costs don't fit within the Testing Group's budget. This money is spent by the programming, design, and marketing staffs.

- ***Appraisal Costs:*** Costs of activities designed to find quality problems, such as code inspections and any type of testing.

---

1 Gryna, F. M. (1988) "Quality Costs" in Juran, J.M. & Gryna, F. M. (1988, 4th Ed.), *Juran's Quality Control Handbook,* McGraw-Hill, page 4.2.

2 Feigenbaum, A.V. (1991, 3rd Ed. Revised), *Total Quality Control*, McGraw-Hill, Chapter 7.

3 Gryna, F. M. "Quality Costs" in Juran, J.M. & Gryna, F. M. (1988, 4th Ed.), *Juran's Quality Control Handbook,* McGraw-Hill, page 4.2. I'm not aware of reliable data on quality costs in software.

4 These are my translations of the ideas for a software development audience. More general, and more complete, definitions are available in Campanella, J. (Ed.) (1990), *Principles of Quality Costs*, ASQC Quality Press, as well as in Juran's and Feigenbaum's works.

Design reviews are part prevention and part appraisal. To the degree that you're looking for errors in the proposed design itself when you do the review, you're doing an appraisal. To the degree that you are looking for ways to strengthen the design, you are doing prevention.

- ***Failure Costs:*** Costs that result from poor quality, such as the cost of fixing bugs and the cost of dealing with customer complaints.

- ***Internal Failure Costs:*** Failure costs that arise before your company supplies its product to the customer. Along with costs of finding and fixing bugs are many internal failure costs borne by groups outside of Product Development. If a bug blocks someone in your company from doing her job, the costs of the wasted time, the missed milestones, and the overtime to get back onto schedule are all internal failure costs.

  For example, if your company sells thousands of copies of the same program, you will probably print several thousand copies of a multi-color box that contains and describes the program. You (your company) will often be able to get a *much* better deal by booking press time in advance. However, if you don't get the artwork to the printer on time, you might have to pay for some or all of that wasted press time anyway, and then you may have to pay additional printing fees and rush charges to get the printing done on the new schedule. This can be an added expense of many thousands of dollars.

  Some programming groups treat user interface errors as low priority, leaving them until the end to fix. This can be a mistake. Marketing staff need pictures of the product's screen long before the program is finished, in order to get the artwork for the box into the printer on time. User interface bugs – the ones that will be fixed later – can make it hard for these staff members to take (or mock up) accurate screen shots. Delays caused by these minor design flaws, or by bugs that block a packaging staff member from creating or printing special reports, can cause the company to miss its printer deadline.

  Including costs like lost opportunity and cost of delays in numerical estimates of the total cost of quality can be controversial. Campanella (1990)[1] doesn't include these in a detailed listing of examples. Gryna (1988)[2] recommends against including costs like these in the published totals because fallout from the controversy over them can kill the entire quality cost accounting effort. I include them here because I sometimes find them very useful, even if it might not make sense to include them in a balance sheet.

- ***External Failure Costs:*** Failure costs that arise after your company supplies the product to the customer, such as customer service costs, or the cost of patching a released product and distributing the patch.

  External failure costs are huge. It is much cheaper to fix problems before shipping the defective product to customers.

---

1 *Principles of Quality Costs*, ASQC Quality Press, Appendix B, "Detailed Description of Quality Cost Elements."

2 "Quality Costs" in Juran, J.M. & Gryna, F. M. (1988, 4th Ed.), *Juran's Quality Control Handbook,* McGraw-Hill, pages 4.9 - 4.12.

Some of these costs must be treated with care. For example, the cost of public relations efforts to soften the publicity effects of bugs is probably not a huge percentage of your company's PR budget. You can't charge the entire PR budget as a quality-related cost. But any money that the PR group has to spend to specifically cope with potentially bad publicity due to bugs is a failure cost.

I've omitted from Figure 1 several additional costs that I don't know how to estimate, and that I suspect are too often too controversial to use. Of these, my two strongest themes are cost of high turnover (people quit over quality-related frustration – this definitely includes sales staff, not just development and support) and cost of lost pride (many people will work less hard, with less care, if they believe that the final product will be low quality no matter what they do.)

- ***Total Cost of Quality:*** The sum of costs: Prevention + Appraisal + Internal Failure + External Failure.

Figure 1. Examples of Quality Costs Associated with Software Products.

Prevention	***Appraisal***
• Staff training • Requirements analysis • Early prototyping • Fault-tolerant design • Defensive programming • Usability analysis • Clear specification • Accurate internal documentation • Evaluation of the reliability of development tools (before buying them) or of other potential components of the product	• Design review • Code inspection • Glass box testing • Black box testing • Training testers • Beta testing • Test automation • Usability testing • Pre-release out-of-box testing by customer service staff
Internal Failure	***External Failure***
• Bug fixes • Regression testing • Wasted in-house user time • Wasted tester time • Wasted writer time • Wasted marketer time • Wasted advertisements[1] • Direct cost of late shipment[2] • Opportunity cost of late shipment	• Technical support calls[3] • Preparation of support answer books • Investigation of customer complaints • Refunds and recalls • Coding / testing of interim bug fix releases • Shipping of updated product • Added expense of supporting multiple versions of the product in the field • PR work to soften drafts of harsh reviews • Lost sales • Lost customer goodwill • Discounts to resellers to encourage them to keep selling the product • Warranty costs • Liability costs • Government investigations[4] • Penalties[5] • All other costs imposed by law

[1] The product is scheduled for release on July 1, so your company arranges (far in advance) for an advertising campaign starting July 10. The product has too many bugs to ship, and is delayed until December. All that advertising money was wasted.

[2] If the product had to be shipped late because of bugs that had to be fixed, the direct cost of late shipment includes the lost sales, whereas the opportunity cost of the late shipment includes the costs of delaying other projects while everyone finished this one.

[3] Note, by the way, that you can reduce external failure costs without improving product quality. To reduce post-sale support costs without increasing customer satisfaction, charge people for support. Switch from a toll-free support line to a toll line, cut your support staff size and you can leave callers on hold for a long time at their expense. This discourages them from calling back. Because these cost reductions don't increase customer satisfaction, the seller's cost of quality is going down, but the customer's is not.

[4] This is the cost of cooperating with a government investigation. Even if your company isn't charged or penalized, you spend money on lawyers, etc.

[5] Some penalties are written into the contract between the software developer and the purchaser, and the developer pays them if the product is late or has specified problems. Other penalties are imposed by law. For example, the developer/publisher of a computer program that prepares United States taxes is liable for penalties to the Internal Revenue Service for errors in tax returns that are caused by bugs or design errors in the program. The publishers are treated like other tax preparers (accountants, tax lawyers, etc.). See *Revenue Ruling 85-189* in *Cumulative Bulletin*, 1985-2, page 341.

What Makes this Approach Powerful?

Over the long term, a project (or corporate) cost accounting system that tracks quality-related costs can be a fundamentally important management tool. This is the path that Juran and Feigenbaum will lead you down, and they and their followers have frequently and eloquently explained the path, the system, and the goal.

I generally work with young, consumer-oriented software companies who don't see TQM programs as their top priority, and therefore my approach is more tactical. There is significant benefit in using the language and insights of quality cost analysis, on a project/product by project/product basis, even in a company that has no interest in Total Quality Management or other formal quality management models.[1]

Here's an example. Suppose that some feature has been designed in a way that you believe will be awkward and annoying for the customer. You raise the issue and the project manager rejects your report as subjective. It's "not a bug." Where do you go if you don't want to drop this issue? One approach is to keep taking it to higher-level managers within product development (or within the company as a whole). But without additional arguments, you'll often keep losing, without making any friends in the process.

Suppose that you change your emphasis instead. Rather than saying that, in your opinion, customers won't be happy, collect some other data:[2]

- ***Ask the writers***: Is this design odd enough that it is causing extra effort to document? Would a simpler design reduce writing time and the number of pages in the manual?
- ***Ask the training staff:*** Are they going to have to spend extra time in class, and to write more supplementary materials because of this design?
- ***Ask Technical Support and Customer Service:*** Will this design increase support costs? Will it take longer to train support staff? Will there be more calls for explanations or help? More complaints? Have customers asked for refunds in previous versions of the product because of features designed like this one?

---

1 I am most definitely not saying that a tactical approach is more practical than an integrated, long-term approach. Gryna notes that there are two common approaches to cost-of-quality programs. One approach involves one-shot studies that help the company identify targets for significant improvement. The other approach incorporates quality cost control into the structure of the business. (Gryna, 1988, in Juran, J. M. & Gryna, F. M. (1988, 4th Ed.), *Juran's Quality Control Handbook,* McGraw-Hill, pages 4.2 onward.) The one-shot, tactical approach can prove the benefit of the more strategic, long-term system to a skeptical company.

2 Be sensitive to how you do this. If you adopt a tone that says that you think the project manager and the programming staff are idiots, you won't enjoy the long-term results.

- ***Check for related problems:*** Is this design having other effects on the reliability of the program? Has it caused other bugs? (Look in the database.) Made the code harder to change? (Ask the programmers.)

- ***Ask the sales staff:*** If you think that this feature is very visible, and visibly wrong, ask whether it will interfere with sales demonstrations, or add to customer resistance.

- ***What about magazine reviews?*** Is this problem likely to be visible enough to be complained about by reviewers? If you think so, check your impression with someone in Marketing or PR.

You won't get cost estimates from everyone, but you might be able to get ballpark estimates from most, along with one or two carefully considered estimates. This is enough to give you a range to present at the next project meeting, or in a follow-up to your original bug report. Notice the difference in your posture:

- You're no longer presenting *your opinion* that the feature is a problem. You're presenting information collected from several parts of the company that demonstrates that this feature's design is a problem.

- You're no longer arguing that the feature should be changed just to improve the quality. No one else in the room can posture and say that you're being "idealistic" whereas a more pragmatic, real-world businessperson wouldn't worry about problems like this one. Instead, *you're* the one making the hard-nosed business argument, "This design is going to cost us $X in failure costs. How much will it cost to fix it?"

- Your estimates are based on information from other stakeholders in this project. If you've fairly represented their views, you'll get support from them, at least to the extent of them saying that you are honestly representing the data you've collected.

Along with arguing about individual bugs, or groups of bugs, this approach opens up opportunities for you (and other non-testers who come to realize the power of your approach) to make business cases on several other types of issues. For example:

- The question of who should do unit testing (the programmers, the testers, or no one) can be phrased and studied as a cost-of-quality issue. The programmers might be more efficient than testers who don't know the code, but the testers might be less expensive per hour than the programmers, and easier to recruit and train, and safer (unlike newly added programmers, new testers can't write new bugs into the code) to add late in the project.

- The depth of the user manual's index is a cost-of-quality issue. An excellent index might cost 35 indexer-minutes per page of the manual (so a 200 page book would take over three person-weeks to index). Trade this cost against the reduction in support calls because people can *find* answers to their questions in the manual.

- The best investment to achieve better quality might be additional training and staffing of the programming group (prevent the bugs rather than find and fix them).

- You (in combination with the Documentation, Marketing, or Customer Service group) might demonstrate that the user interface must be fixed and frozen sooner because of the impact of late changes on the costs of developing documentation, packaging, marketing collaterals, training materials, and support materials.

Implementation Risks

Gryna (1988)[1] and Juran & Gryna (1980)[2] point out several problems that have caused cost-of-quality approaches to fail. I'll mention two of the main ones here.

First, it's unwise to try to achieve too much, too fast. For example, don't try to apply a quality cost system to every project until you've applied it successfully to one project. And don't try to measure all of the costs, because you probably can't.[3]

Second, beware of insisting on controversial costs. Gryna (1988)[4] points out several types of costs that other managers might challenge as not being quality-related. If you include these costs in your totals (such as total cost of quality), some readers will believe that you a padding these totals, to achieve a more dramatic effect. Gryna's advice is to not include them. This is usually wise advice, but it can lead you to underestimate your customer's probable dissatisfaction with your product. As we see in the next section, down that road lies LawyerLand.

The Dark Side of Quality Cost Analysis

Quality Cost Analysis looks at the company's costs, not the customer's costs. The manufacturer and seller are definitely not the only people who suffer quality-related costs. The customer suffers quality-related costs too. If a manufacturer sells a bad product, the customer faces significant expenses in dealing with that bad product.

---

1 in Juran, J. M. & Gryna, F. M. (1988, 4th Ed.), *Juran's Quality Control Handbook,* McGraw-Hill, pages 4.27-4.28.

2 *Quality Planning and Analysis* (2nd Ed.), McGraw-Hill, pages 30-32. Also, see Brown, M. G., Hitchcock, D. E. & Willard, M. L. (1994), W*hy TQM Fails and What To Do About It*, Irwin Professional Publishing.

3 As he says in Juran, J.M. (1992), *Juran on Quality by Design*, The Free Press, p. 119, Costs of poor quality "are huge, but the amounts are not known with precision. In most companies the accounting system provides only a minority of the information needed to quantify this cost of poor quality. It takes a great deal of time and effort to extend the accounting system so as to provide full coverage. Most companies have concluded that such an effort is not cost effective. [¶] What can be done is to fill the gap by *estimates*, which provide managers with approximate information as to the total cost of poor quality and as to where the major areas of concentration are."

4 "Quality Costs" in Juran, J.M. & Gryna, F. M. (1988, 4th Ed.), *Juran's Quality Control Handbook,* McGraw-Hill, pages 4.9 - 4.12.

The Ford Pinto litigation provided the most famous example of a quality cost analysis that evaluated company costs without considering customers' costs from the customers' viewpoint. Among the documents produced in these cases was the Grush-Saunby report, which looked at costs associated with fuel tank integrity. The key calculations appeared in Table 3 of the report:[1]

Benefits and Costs Relating to Fuel Leakage Associated with the Static Rollover Test Portion of FMVSS 208
Benefits Savings – 180 burn deaths, 180 serious burn injuries, 2100 burned vehicles Unit Cost -- $200,000 per death, $67,000 per injury, $700 per vehicle Total Benefit – 180 x ($200,000) + 180 x ($67,000) + 2100 x ($700) = $49.5 million.
Costs Sales – 11 million cars, 1.5 million light trucks. Unit Cost -- $11 per car, $11 per truck Total Cost – 11,000,000 x ($11) + 1,500,000 x ($11) = $137 million.

In other words, it looked cheaper to pay an average of $200,000 per death in lawsuit costs than to pay $11 per car to prevent fuel tank explosions. Ultimately, the lawsuit losses were much higher.[2]

This kind of analysis didn't go away with the Pinto. For example, in the more recent case of *General Motors Corp. v. Johnston* (1992),[3] a PROM controlled the fuel injector in a pickup truck. The truck stalled because of a defect in the PROM and in the ensuing accident, Johnston's seven-year old grandchild was killed. The Alabama Supreme Court justified an award of $7.5 million in punitive damages against GM by noting that GM "saved approximately $42,000,000 by not having a recall or otherwise notifying its purchasers of the problem related to the PROM."

Most software failures don't lead to deaths. Most software projects involve conscious tradeoffs among several factors, including cost, time to completion, richness of the feature set, and reliability. There is nothing wrong with doing this type of business tradeoff, consciously and explicitly, unless you fail to take into account the fact that some of the problems that you leave in the product might cost your customers much, much more than they cost your company. Figure 2 lists some of the external failure costs that are borne by customers, rather than by the company.

---

1 This table is published in Keeton, W. P., Owen, D.G., Montgomery, J. E., & Green, M.D. (1989, 2nd Ed.) *Products Liability and Safety, Cases and Materials,* Foundation Press, page 841 and Posner, R.A. (1982) *Tort Law: Cases and Economic Analysis*, Little Brown & Co., page 225.

2 *Grimshaw v. Ford Motor Co*. (1981), (California Court of Appeal), *California Reporter*, volume 174, page 348.

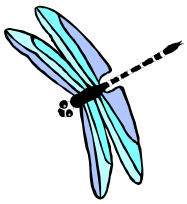
3 *Southern Reporter*, 2nd Series, volume 592, pages 1054 and 1061.

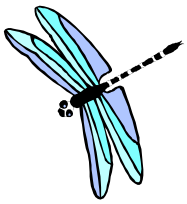
Figure 2. Comparison of External Failure Costs Borne by the Buyer and the Seller

Seller: external failure costs	***Customer: failure costs***
These are the types of costs absorbed by the seller that releases a defective product.	These are the types of costs absorbed by the customer who buys a defective product.
• Technical support calls • Preparation of support answer books • Investigation of customer complaints • Refunds and recalls • Coding / testing of interim bug fix releases • Shipping of updated product • Added expense of supporting multiple versions of the product in the field • PR work to soften drafts of harsh reviews • Lost sales • Lost customer goodwill • Discounts to resellers to encourage them to keep selling the product • Warranty costs • Liability costs • Government investigations • Penalties • All other costs imposed by law	• Wasted time • Lost data • Lost business • Embarrassment • Frustrated employees quit • Demos or presentations to potential customers fail because of the software • Failure when attempting other tasks that can only be done once • Cost of replacing product • Cost of reconfiguring the system • Cost of recovery software • Cost of tech support • Injury / death

The point of quality-related litigation is to transfer some of the costs borne by a cheated or injured customer back to the maker or seller of the defective product. The well-publicized cases are for disastrous personal injuries, but there are plenty of cases against computer companies and software companies for breach of contract, breach of warranty, fraud, etc.

The problem of cost-of-quality analysis is that it sets us up to underestimate our litigation and customer dissatisfaction risks. We think, when we have estimated the total cost of quality associated with a project, that we have done a fairly complete analysis. But if we don't take customers' external failure costs into account at some point, we can be surprised by huge increased costs (lawsuits) over decisions that we thought, in our incomplete analyses, were safe and reasonable.





What About The Objection That It Is Not Really A Bug?

Really, it's a feature.

Or, at least, it's not a problem for my release so I don't have to fix it.

It won't matter until we ship it to Germany. Let them fix it.

Software Errors: What is Quality?

Here are some of the traditional definitions:

- Fitness for use (Dr. Joseph M. Juran)
- The totality of features and characteristics of a product that bear on its ability to satisfy a given need (ASQ)
- Conformance with requirements (Philip Cosby)
- The total composite product and service characteristics of marketing, engineering, manufacturing and maintenance through which the product and service in use will meet expectations of the customer (Armand V. Feigenbaum)

Note the absence of "conforms to specifications."

Software Errors: What is Quality?

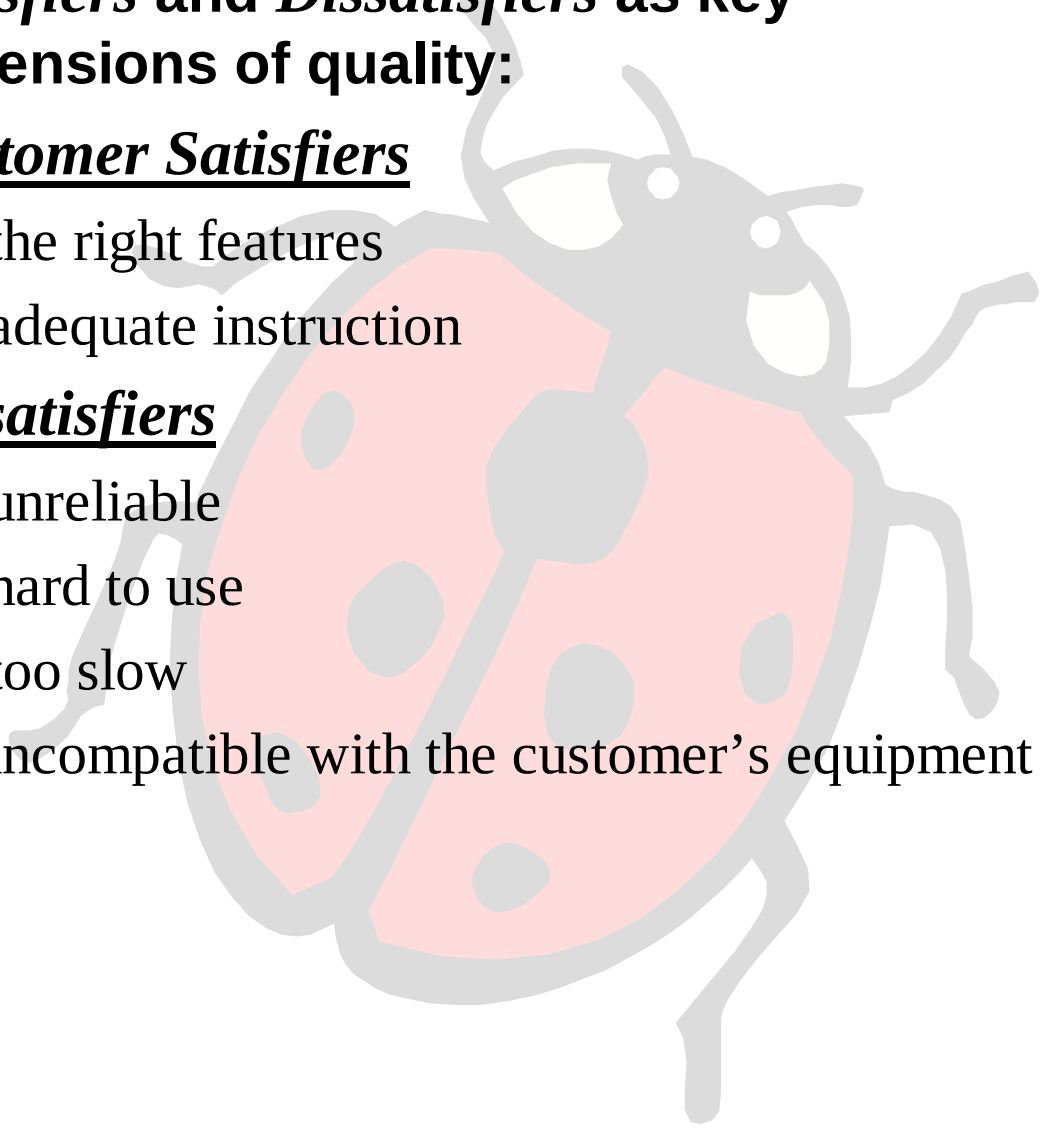
Juran distinguishes between *Customer Satisfiers* and *Dissatisfiers* as key dimensions of quality:

Customer Satisfiers

- the right features
- adequate instruction

Dissatisfiers

- unreliable
- hard to use
- too slow
- incompatible with the customer's equipment



Software Errors: What Should We Report?

I like Gerald Weinberg's definition:

Quality is value to some person

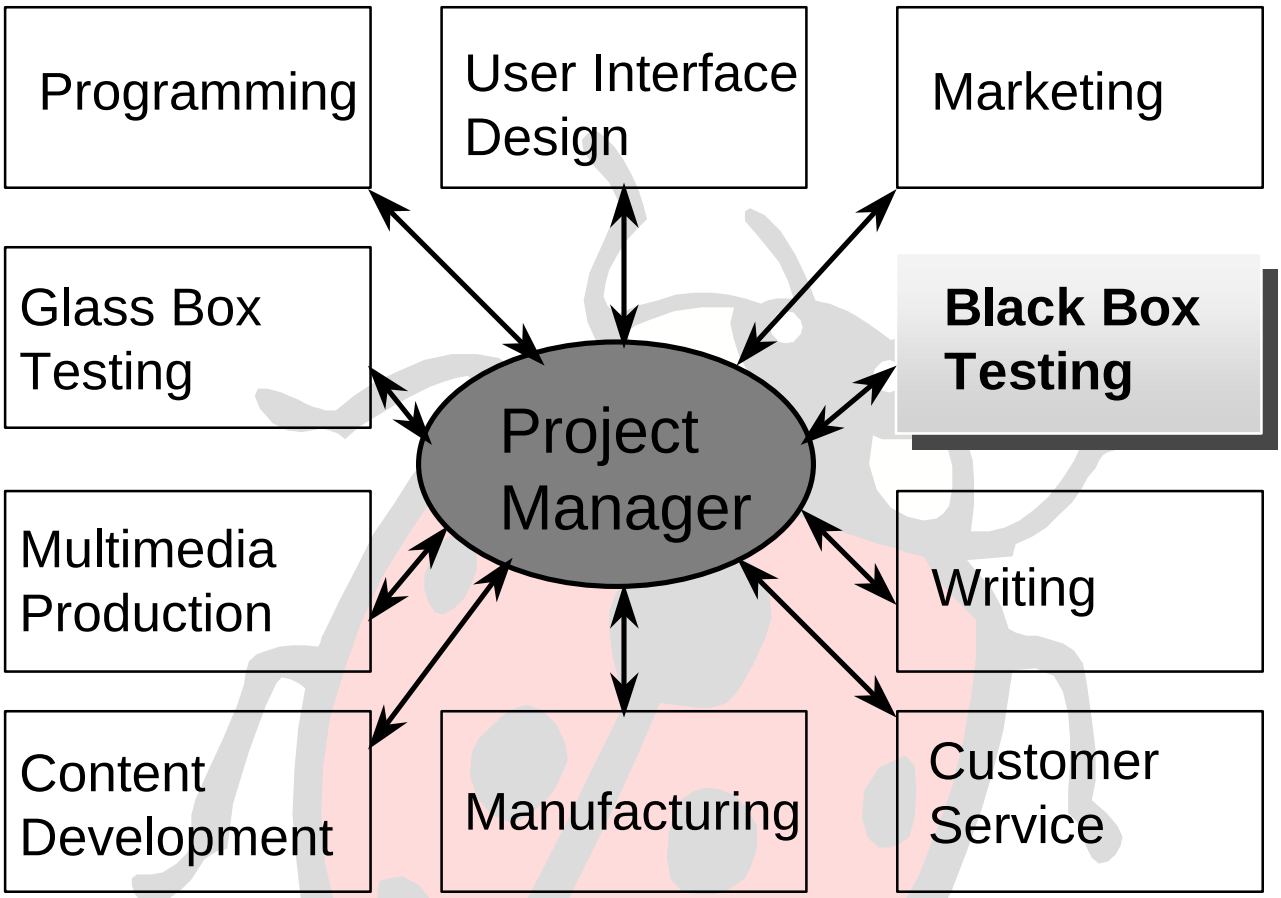
But consider the implication:

- **It's appropriate to report any deviation from high quality as a software error.**
- **Therefore many issues will be reported that will be errors to some and non-errors to others.**

Glen Myers' definition:

- A software error is present when the program does not do what its user reasonably expects it to do.

Quality is Multidimensional



When you sit in a project team meeting, discussing a bug, a new feature, or some other issue in the project, you must understand that each person in the room has a different vision of what a “quality” product would be. Fixing bugs is just one issue. The next slide gives some examples.

Quality is Multidimensional: Different People, Different Visions

Localization Manager: *A good product is easy to translate and to modify to make it suitable for another country and culture. Few expereienced localization managers would consider acceptable a product that must be recompiled or relinked to be localized.*

Tech Writers: *A high quality program is easily explainable. Aspects of the design that are confusing, unnecessarily inconsistent, or hard to describe are marks of bad quality.*

Marketing: *Customer satisfiers are the things that drive people to buy the product and to tell their friends about it. A Marketing Manager who is trying to add new features to the product generally believes that he is trying to improve the product.*

Customer Service: *Good products are supportable. They have been designed to help people solve their own problems or to get help quickly.*

Programmers: *Great code is maintainable, well documented, easy to understand, well organized, fast and compact.*

Software Errors: Why are there Errors?

New testers often conclude that the programmers on their project are incompetent or unprofessional.

- This is counterproductive. It leads to infighting instead of communication, and it leads to squabbling over bugs instead of research and bug fixing.
- And as we saw when we discussed private bug rates, programmers actually find and fix the large majority of their own bugs.
- Bugs come into the code for many reasons. It's worth considering some common systematic (as distinct from poor individual performance) factors. You will learn to vary your strategic approaches as you learn your companies' weaknesses.

Bugs come into the code for many reasons:

- The major cause of error is that programmers deal with tasks that aren't fully understood or fully defined. This is said in many different ways. For example:
  - » Tom Gilb and Dick Bender quote industry-summary statistics that 80% of the errors, or 80% of the effort required to fix the errors, are caused by bad requirements;
  - » Roger Sherman recently summarized research at Microsoft that the most common underlying issue in bug reports involved a need for new code.

  If you graduated from a Computer Science program, how much training did you have in task analysis? Requirements definition? Usability analysis? Negotiation and clear communication of negotiated agreements? Not much? Hmmmm . . . .
- Some companies drive their programmers too hard. They don't have enough time to design, bulletproof, or test their code. Another Sherman quote: "Bad schedules are responsible for most quality problems."
- Late design changes result in last minute code changes, which are likely to have errors.
- Some third-party components introduce bugs. Your program might rely on a large suite of small components that display a specific type of object, filter data in a special way, drive a specific printer, etc. Many of these tools, bought from tool vendors or hardware vendors, are surprisingly buggy. Others work, but they aren't fully compatible with common test automation tools.

Software Errors: Why are there Errors?

- Some programs or tasks are inherently complex. Boris Beizer talks perceptively about the locality problem in software. Think about an underlying bug, and then about the symptoms caused by the bug. When symptoms appear, there is no assurance that they will be close in time, space, or severity to the underlying bug. They may appear much later, or when working with a different part of the program, and they may seem much more or much less serious than the bug.
- Failure to use source control tools creates characteristic bugs. For example, if a bug goes away, comes back, goes away, comes back, goes away, comes back, then ask how the programming staff makes sure it's linking the most recent version of each module when it builds a version for you to test.
- Some programmers (some platforms) work with poor tools. Weak compilers, style checkers, debuggers, profilers, etc. make it too easy to get bugs or too hard to find bugs.
- Similarly, some third party hardware, or its drivers, are non-standard and don't respond properly to standard system calls. Incompatibility with hardware is often cited as the largest single source of customer complaints into technical support groups.
- When one programmer tries to fix bugs, or otherwise modify another programmer's code, there is a lot of room for miscommunication and error.
- And, sometimes people just make mistakes.

Quality: Family Drug Store v. Gulf States Computer

(563 So.2d 1324, Louisiana Court of Appeal, 1990). *The basic holding of this case is that a computer program that is honestly marketed can be unacceptably awkward to use without imposing liability on the seller.*

Two pharmacists bought a computer program known as the Medical Supply System from Gulf States. After they realized what they had bought, they asked for, and then sued for, a refund. Here were some of the problems of the system:

"1 all data had to be printed out, and could not be viewed on the monitor;

2 the information on the monitor would appear in code;

3 numerical codes were needed in order to open a new patient file

4 the system was unable to scroll."

The court found that the seller had not in any way misrepresented the system, and that it was not useless even though it was awkward to use. Further, the price of the software was about $2500 compared to $10,000 for other packages. *The plaintiffs had gotten what they'd paid for.*

Cem Kaner, Ph.D., J.D. P.O. Box 1200
Attorney at Law Santa Clara, CA 95052
kaner@kaner.com 408-244-7000

What is a Software Defect?

One discussion that plagues development groups is how serious a bug is. Can we call it a feature? Does it have to be fixed now or can it wait until the next release? What are the consequences if we ship it?

This article looks at this issue from a different angle:

> *How should the law determine whether a bug is serious enough that the customer should be entitled to cancel the contract, return the software, and demand a refund?*

I've been asked to write the first draft of language to include in a new law. More than any other piece of the new statute, this section will affect the day-to-day interactions within product development groups. So, I'm asking you to review this proposal and send me your comments, at kaner@kaner.com.

Background: UCC Draft Article 2B

A committee of the National Conference of Commissioners on Uniform State Laws (NCCUSL) is drafting a new Article (2B) for the Uniform Commercial Code (UCC). UCC Article 2 is the Law of Sales, which currently governs the licensing and sales of most software products. In the future, Article 2B will govern the sale and licensing of all software products (and of most other types of information products, such as cable TV).

For the last year, I've been attending meetings of the Article 2B Committee, arguing that customers need more protection than the drafts of Article 2B have been providing. [i] One issue that keeps coming up is the severity of bugs.

A software defect is a "material breach" of the contract for sale or license of the software if it is so serious that the customer can justifiably demand a fix or can cancel the contract, return the software, and demand a refund. The American Bar Association's Committee on Computer Programs calls these *Material Bugs*.[ii] I'll use the same phrase.

---

[i] Some readers of this magazine might have read my previous paper on Article 2B: Kaner, C. "Uniform Commercial Code Article 2B: A New Law of Software Quality," *Software QA*, Volume 3, #2, 1996, p. 10.

[ii] American Bar Association, Section on Patent, Trademark, & Copyright Law, Committee on Computer Programs *Model Software Licensing Agreement*, 1992.

If a defect is not "material" then the customer is stuck with the program and is probably entitled to, at most, a partial refund.

How should we decide whether a bug is serious enough to be called "material"?

The July (and previous drafts) of Article 2B

Here's the proposed definition of a material breach of the software contract in July, 1996 (and in several previous drafts).

> 2B-109 (July) (b) A breach is material if the contract so provides or, in the absence of express contractual terms, if the circumstances, intent of the parties, language of the contract, and character of the breach indicate that the breach caused or may cause substantial harm to the interests of the aggrieved party, or if it meets the conditions of subsection (c) or (d).
>
> (c) A breach is material if it involves:
>
> (1) knowing or grossly negligent disclosure or use of confidential information of the aggrieved party not justified by the license;
>
> (2) knowing infringement of the aggrieved party's intellectual property rights not authorized by the terms of the license and occurring over more than a brief period; or
>
> (3) an uncured, substantial failure to pay a license fee when due which is not justified by an existing, colorable dispute about whether payment is due.
>
> (d) A material breach occurs if the aggregate effect of the nonmaterial breaches by the same party satisfy the standards for materiality.

Look closely at (c), which defines a material breach by listing examples. All of these protect the publisher from breaches of the contract by the customer. For example, the customer is in trouble if s/he (1) discloses the publisher's confidential information, (2) pirates the software, or (3) doesn't pay for the program.

None of these helps the customer argue that a given bug is material. In fact, because the statute lists a series of things that are "material," a common rule of statutory interpretation would suggest to lawyers and judges that nothing that is not in the list can be material. So, maybe no bug can be so important that it materially breaches of the contract.

The September and November Drafts

The September, 1996 draft added a further clause to the list. A breach is also material if it involves:

> a failure to perform in a manner consistent with express performance standards;

In September, I asked Ray Nimmer (the Reporter of the Committee and the lead author of the Article 2B drafts) what this means. What is an "express performance standard?" Is it just a precise, verifiable statement about the program that the seller makes to the customer? Can the customer get a refund if the program fails to match the written documentation

He said, no, that's not what he meant. He was referring to the specific promises that were contained in a negotiated contract, that said that the program *must* do specified things or have specified characteristics.

So I said that this is unfair. What about customers who don't have negotiated contracts that specify all the right things? What about mass-market customers? (Mass-market software is made available to the general public, in a way that is not customized for individual customers. For example, Microsoft Word is mass-market software.) If the program doesn't work, don't these customers have any recourse?

Ray said that this is a genuine problem. But he doesn't know how to solve it. *How should we define material defect when the bug is not a direct nonconformity with an important part of the specification?*

Then he said to me that I was the person who kept raising these issues, so maybe I could take a crack at drafting the definition.

And I promised that I would. But I need your help.

How Should We Define a Serious Defect?

This paper provides a first, working draft of a definition of a material bug.

Failure to conform to specifications is a common theme in the legal books, but many of the software development contracts provide vague, incomplete specifications that will change over time without being updated in the contract itself. Whether the specification or user documentation addresses the following issues, all of them should be considered material bugs, shouldn't they?

- the product doesn't work at all
- the disks are blank
- the product caused substantial harm to the property or the business of the licensee
- the licensor supplied a product that lack promised or advertised features or capabilities
- the software deprives the customer of a key benefit that she reasonably expected (for example, anyone would expect a word processor to be able to print and it's unreasonable if a particular one can't provide that benefit.)
- the customer spent so much time, effort, and money dealing with a bug that the customer's costs exceeded the cost of the product. (If you think that this isn't a serious enough loss to the customer, what if the customer's losses run at ten times the cost of the software, or one hundred times the cost? At some point, the amount of the losses caused by the software become excessive, doesn't it?)

In the course of writing this article, I reviewed a few dozen books, articles and standards on software quality and software defects, trying to understand how different segments of the software industry deal with errors. I also reviewed many court cases and contract books, trying to understand how attorneys currently deal with software errors. These materials provided lots of data, but not a solid framework. I think that the framework provided here is sensible, useful, fair, and workable for software developers as well as for legal practitioners, but it needs review.

Reflecting the Relationships Between Licensor and Licensee

The licensor is analogous to the seller in a traditional sale. Under Article 2B, what is sold in the typical software transaction is a license to use the software, rather than a copy of the software. The licensee is the customer, who buys the license.

I think there are four common classes of software transaction:

1. ***The customer writes the specifications and requirements and asks the developer to write a program as specified.***

   In my view, the software developer meets its [i] obligations if it writes a program that meets the specifications. If the specs say "2+2=5" then the program does not breach the software contract if it generates the wrong answer (5) whenever it adds 2+2. Let the specifier beware.

2. ***The developer writes the specifications and requirements, in preparation for custom development, but the customer is sophisticated.***

   If the customer is a computer expert, then it is able to review the requirements and specifications just as well as the staff of the developer. The customer is also probably in a better position to understand its own requirements than the developer. Therefore it is reasonable to hold this customer accountable for reviewing the specifications.

   Article 2 of the current Uniform Commercial Code defines a "merchant" as follows:

   2-104 (1) "Merchant" means a person who deals in goods of the kind or otherwise by his occupation holds himself out as having knowledge or skill peculiar to the practices or goods involved in the transaction or to whom such knowledge or skill may be attributed by his employment of an agent or broker or other intermediary who by his occupation holds himself out as having such knowledge or skill.

   2-104(3) "Between merchants" means in any transaction with respect to which both parties are chargeable with the knowledge or skill of merchants.

   Article 2B uses essentially the same definition:

   2B-102 (26) "Merchant" means a person that deals in information of the kind, a person that by occupation purports to have knowledge or skill peculiar to the practices or information involved in the transaction, or a person to which knowledge or skill may be attributed by the person's employment of an agent or broker or other intermediary that purports to have the knowledge or skill.

   When Microsoft buys Apple computers, it is a merchant. When a large, local hospital buys a bunch of Apples, it might be a big business, but it is not a merchant.

---

[i] In legal writing, it is common to use the word "it" instead of "he" or "she" whenever the being (here, the software developer) is likely to be a corporation rather than an individual human.

3) ***The developer writes the specifications and requirements, in preparation for custom development, but the customer is not sophisticated.***

When doctors, dentists, insurance brokers, small grocery store owners, and other small business people buy software, they have no clue how to specify the software, no clue how to evaluate a requirements document, no clue how to test the software, and no clue how to cost-effectively find a consultant who has these skills. In common computer parlance, these customers are called *Clueless.*

In UCC parlance, these customers are *non-merchants.*

These customers rely on the knowledge and experience of the developer. If the developer makes errors in defining the requirements or the specifications, which result in serious errors in the operation of the program, this is the developer's bug, not the customer's.

4) ***The developer writes a mass-market product. The customer has no input into the design or development of the product.***

In the mass-market case, design errors belong to the developer, not the customer. Internal specifications that were used during development are largely irrelevant to the customer. The end product works in a reasonable way, as advertised and as documented, or it does not.

The Proposed Statutory Language

This proposal modifies Section 2B-108 of Article 2B. I am a novice at drafting statutes. The language will be cleaned up during the Article 2B review process. The proposal expresses my sense of the fundamental differences between these transactions.

SECTION 2B-108. BREACH OF CONTRACT.

(a) Whether a party is in breach of contract is determined by the terms of the agreement and by this article. Breach occurs if a party fails to perform an obligation timely or exceeds a contractual limitation.

(b) A breach of contract is material if the contract so provides. In the absence of express contractual terms, a breach is material if the circumstances, including the language of the agreement, expectations of the parties, and character of the breach, indicate that the breach caused or may cause substantial harm to the interests of the aggrieved party, that the injured party will be substantially deprived of the benefit it reasonably expected under the contract, or that the breach meets the conditions of subsection (c), (d), (e), (f) or (g).

(c) If the licensee provides the specification documents that are incorporated in the contract, then a breach is material if:

(i) the software fails to perform in conformance with and in the time required by express performance standards or specifications;

(ii) the software fails to perform in conformance with the specifications and this failure either deprives the licensee of a significant benefit of the product or results in costs to the licensee that exceed the price paid for the software;

(iii) where the specifications are silent, the software's performance is unreasonable and it results in costs to the licensee that exceed the price paid for the software. The licensee has the burden of demonstrating that a reasonable licensor would consider the software's performance to be unreasonable.

(d) If the contract is between merchants, and it contains specification documents, then a breach is material if:

(i) the software fails to perform in conformance with and in the time required by express performance standards or specifications;

(ii) the software fails to perform in conformance with the specifications and this failure either deprives the licensee of a significant benefit of the product or results in costs to the licensee that exceed the price paid for the software;

(iii) where the specifications are silent, the software's performance is unreasonable and it results in costs to the licensee that exceed the price paid for the software. The licensee has the burden of demonstrating that a reasonable licensor would consider the software's performance to be unreasonable.

(e) If the contract is not between merchants, and the licensor provides the specification documents that are incorporated in the contract, then a breach is material if:

(i) the software fails to perform in conformance with and in the time required by express performance standards or specifications;

(ii) the software fails to perform in conformance with the specifications and this failure either deprives the licensee of a significant benefit of the product or results in costs to the licensee that exceed the price paid for the software;

(iii) the software fails to perform in conformance with the end user documentation or other documentation delivered to the licensee and this failure either deprives the licensee of a significant benefit of the product or results in costs to the licensee that exceed the price paid for the software;

(iv) where the specifications and other documentation are silent, the software's performance is unreasonable and as a result, it either deprives the licensee of a significant benefit of the product or it results in costs to the customer that exceed the price paid for the software. The licensee has the burden of demonstrating that a reasonable person would consider the software's performance to be unreasonable.

(f) If the contract is for a mass-market license, then a breach is material if:

(i) the software fails to perform in conformance with the end user documentation or other documentation delivered to the licensee and this failure either deprives the licensee of a significant benefit of the product or results in costs to the customer that exceed the price paid for the software;

(ii) where the documentation is silent, the software's performance is unreasonable and as a result, it either deprives the licensee of a significant benefit of the product or it results in costs to the licensee that exceed the price paid for the software. The licensee has the burden of demonstrating that a reasonable person would consider the software's performance to be unreasonable.

(g) A material breach of contract occurs if the cumulative effect of nonmaterial breaches by the same party satisfies the standards for materiality.

(h) If there is a breach of contract, whether or not material, the aggrieved party is entitled to the remedies provided for in this article and the agreement.

What Happens from Here?

By the time you read this proposal, I will have circulated it to the Article 2B Drafting Committee. They'll probably consider it at the January 10-12 Drafting Committee meeting at the Sofitel Hotel in Redword City, California. The next meeting of the Committee will be in Atlanta from February 21 to 23, 1997. I will compile comments that people send me, and will summarize them for this meeting. You can also attend either meeting yourself. Few of the attendees are non-lawyers, but you are welcome to speak if you have something informative to say.

This process will continue for a few more months (four meetings are scheduled in 1997), probably resulting in legislation that is introduced in the state legislatures in 1998. Whether you or I participate in this process or not, the result will include rules that govern software quality, laying out the ground rules under which we decide whether bugs are features and whether they need to be fixed. We can influence the process.

To read the latest draft of Article 2B, and to send comments directly to Ray Nimmer, the Drafting Committee's Reporter, visit the Article 2B home page at www.law.uh.edu/ucc2b.

APPENDIX

The following notes weren't included in the SQA article but did appear in the memo actually considered by the UCC Drafting Committee.

How Should We Define a Serious Defect?

Failure to conform to specifications is a common theme in legal books, but many of the software development contracts provide vague, incomplete specifications that will change over time without being updated in the contract itself. Discussions within the software development community consistently recognize that most failures in commercial software products are due to errors in the specifications or requirements. A widely used number is that 80% of the money spent fixing or dealing with software problems can be traced back to requirements errors.

As a result, texts that focus on software errors don't limit themselves to failure to meet a specification (this type of failure is called *nonconformance*). Here are some examples from well respected texts in the field:

IEEE (1989), *IEEE Standard Dictionary of Measures to Produce Reliable Software*, ANSI/IEEE Standard 982.1-1988, p. 13:

> Defect: A product anomaly. Examples include such things as (1) omissions and imperfections found during early life cycle phases and (2) faults contained in software sufficiently mature for test or operation. See also *fault*.

IEEE (1994), *IEEE Standard Classification for Software Anomalies*, IEEE Standard 1044-1993, p. 3.

> Anomaly: Any condition that deviates from expectations based on requirements specifications, design documents, user documents, standards, etc., or from someone's perceptions or experiences. Anomalies may be found during, but not limited to, the review, test, analysis, compilation, or use of software products or applicable documentation.

Grady, Robert B. & Caswell, Deborah, L. (1987) *Software Metrics: Establishing a Company-Wide Program*. PTR Prentice-Hall, p. 78

> A defect is any flaw in the specification, design, or implementation of a product. . . . If a flaw could not possibly have been detected, or if it could have been detected and would not have been corrected then it is an enhancement. Defects do not include typographical or grammatical errors in engineering documentation.

Ishikawa, Kaoru (translated by David J. Lu) (1985) *What is Total Quality Control? The Japanese Way*, Prentice-Hall. (Ishikawa is the leading Japanese quality control theoriest):

On page 46 ff. he explains why he doesn't trust quality as measured in terms of compliance with standards and specifications. The problem is that these are not true measures of the quality of the product. He works through an excellent example dealing with a role of newsprint. The true measure of quality is whether the paper rips while on the rotary press. Published standards, in terms of such things as tensile strength, provide only secondary measures of how the product will perform in the field. Continuing, on page 56, "There are no standards– whether they be national, international, or company-wide– that are perfect. Usually standards contain some inherent defects. Consumer requirements also change continuously, demanding higher quality year after year. Standards that were adequate when they were first established, quickly become obsolete. [¶]We engage in QC to satisfy customer requirements"

Jones, Capers (1991), *Applied Software Measurement*, McGraw-Hill, page 273.

A software defect is simply a bug which if not removed would cause a program or system to fail or to produce incorrect results. Note: the very common idea that a defect is a failure to adhere to some user requirements is unsatisfactory because it offers no way to measure requirements defects themselves, which constitute one of the larger categories of software error.

Mundel, August, B. (1991) *Ethics in Quality*, ASQC Quality Press, p. 164.

Any variation from the specifications is a nonconformity . . . . There is a group of nonconformities which represent serious threats to the welfare of users and bystanders. These nonconformities are called *defects*, and they not only can cause injury but may also result in the manufacturers, designers, or sellers being sued under the product liability laws. There are also a class of defects called *design defects* which can be responsible for customer dissatisfaction, loss, injury or death. Despite the fact that all of the product conforms to the design, the product is faulty and is not properly designed.

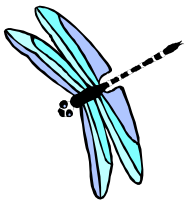
Myers, Glenford J. (1976), *Software Reliability: Principles & Practices*, John Wiley & Sons, pp. 4-6. This is one of the seminal books in the software testing / quality control literature.

One common definition is that a software occurs when the software does not perform according to its specifications. This definition has one fundamental flaw: it tacitly assumes that the specifications are correct. This is rarely, if ever, a valid assumption: one of the major sources of errors is the writing of specifications. . . . [¶] A second common definition is that an error occurs when the software does not perform according to its specifications providing that it is used within its design limits. This definition is actually poorer than the first one....

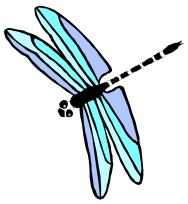
> [¶] A third possible definition is that an error occurs when the software does not behave according to the official documentation or publications supplied to the user. Unfortunately, this definition also has several flaws. There exists the possibility that the software *does* behave according to the official publications but errors are present because both the software and the publications are in error. A second problem occurs because of the tendency of user publications to describe only the manual for a time-sharing system that states, "To enter a new command press the attention key once and type the command." Suppose that a user presses the attention key twice by accident and the software system fails because its designers did not plan for this condition. The system obviously contains an error, but we cannot really state that the system is not behaving according to its publications. [¶] The last definition that is sometimes used defines an error as a failure of the software to perform according to the original contract or documentation. Although this definition is an improvement over the previous three, it still has several flaws . . . written user requirements are rarely detailed enough to describe the desired behavior of the software under all possible circumstances. [¶] There is, however, a reasonable definition of a software error that solves the aforementioned problems: *A software error is present when the software does not do what the user reasonably expects it to do."*

Roetzheim, William H. (1991) *Developing Software to Government Standards*, Prentice-Hall, p. 146

> "Software defects can be divided into four broad categories: (1) requirements defects, (2) design defects, (3) code defects, and (4) documentation defects." See also Dunn, Robert (1984) *Software Defect Removal*, McGraw-Hill, p. 6-7 for the same distinctions.



Bug Advocacy



Writing the Bug Report

Reporting Errors

As soon as you run into a problem in the software, fill out a Problem Report form.

In the well written report, you:

- Explain how to reproduce the problem.
- Analyze the error so you can describe it in a minimum number of steps.
- Include all the steps.
- Make the report easy to understand.
- Keep your tone neutral and non-antagonistic.
- Keep it simple: one bug per report.
- If a sample test file is absolutely essential to reproducing a problem, reference it and attach the test file to the report.
- *To the extent that you have time, describe the dimensions of the bug and characterize it. Describe what events are and are not relevant to the bug. And what the results are (any characteristics of the failure) and how they varied across tests.*

What Are You Reporting?

When you run into a defect, you aren't looking at the program, you are looking at the program's misbehavior.

- An *error* (or *fault)* is a design flaw or a deviation from a desired or intended state.
- An error won't yield a failure without the *conditions* that trigger it. Example, if the program yields 2+2=5 on the tenth time you use it, you won't see the error before or after the tenth use.
- The *failure* is the program's actual incorrect or missing behavior under the error-triggering conditions.

What you report are the conditions and the failure.

The Problem Report Form:

A typical form includes many of the following fields

- ***Problem report number:*** must be unique
- ***Reported by:*** original reporter's name. Some forms add an editor's name.
- ***Date reported:*** date of initial report
- ***Program (or component) name:*** the visible item under test
- ***Release number:*** like Release 2.0
- ***Version (build) identifier:*** like version C or version 20000802a
- ***Configuration(s):*** h/w and s/w configs under which the bug was found and replicated
- ***Report type:*** e.g. coding error, design issue, documentation mismatch, suggestion, query
- ***Can reproduce:*** yes / no / sometimes / unknown. (Unknown can arise, for example, when the repro configuration is at a customer site and not available to the lab.)
- ***Severity:*** assigned by tester. Some variation on small / medium / large
- ***Priority:*** assigned by programmer/project manager
- ***Customer impact:*** often left blank. When used, typically filled in by tech support or someone else predicting actual customer reaction (such as support cost or sales impact)
- ***Problem summary:*** 1-line summary of the problem
- ***Key words:*** use these for searching later, anyone can ***add*** to key words at any time
- ***Problem description and how to reproduce it:*** step by step repro description
- ***Suggested fix:*** leave it blank unless you have something useful to say
- ***Assigned to:*** typically used by project manager
- ***Comments:*** free-form, arbitrarily long field. Programmer, testers and others have an ongoing discussion of the bug, repro conditions, etc., here until it is fixed. Closing comments (why a deferral is OK, for example) go here.
- ***Status:*** Tester fills this in. Open / closed / dumpster—see prev slides on dumpster.
- ***Resolution:*** pending / fixed / deferred / as designed / can't repro / withdrawn by tester
- ***Resolution version:*** build identifier
- ***Resolved by:*** programmer, project manager, tester (if withdrawn by tester), etc.
- ***Resolution tested by:*** originating tester, or a tester if originator was a non-tester
- ***Change history:*** datestamped list of all cha ges to the record, including name and fields changed.

Important Parts of the Report Problem Summary

This one-line description of the problem is the most important part of the report.

- The project manager will use it in when reviewing the list of bugs that haven't been fixed.
- Executives will read it when reviewing the list of bugs that won't be fixed. They might only spend additional time on bugs with "interesting" summaries.

The ideal summary gives the reader enough information to help her decide whether to ask for more information. It should include:

- A brief description that is specific enough that the reader can visualize the failure.
- A brief indication of the limits or dependencies of the bug (how narrow or broad are the circumstances involved in this bug)?
- Some other indication of the severity (not a rating but helping the reader envision the consequences of the bug.)

Important Parts of the Report Can You Reproduce The Problem?

You may not see this on your form, but you should always provide this information.

- Never say it's reproducible unless you have recreated the bug. (Always try to recreate the bug before writing the report.)
- If you've tried and tried but you can't recreate the bug, say "No". Then explain what steps you tried in your attempt to recreate it.
- If the bug appears sporadically and you don't yet know why, say "Sometimes" and explain.
- You may not be able to try to replicate some bugs. Example: customer-reported bugs where the setup is too hard to recreate.

The following policy is not uncommon:

- **If the tester says that a bug is reproducible and the programmer says it is not, then the tester has to recreate it in the presence of the programmer.**

Important Parts of the Report Description; How to Reproduce It.

- First, describe the problem. What's the bug? Don't rely on the summary to do this -- some reports will print this field without the summary.
- Next, go through the steps that you use to recreate this bug.
  - » Start from a known place (e.g. boot the program) and
  - » then describe each step until you hit the bug. NUMBER THE STEPS. Take it one step at a time.
- Describe the erroneous behavior and, if necessary, explain what should have happened. (Why is this a bug? Be clear.)
- List the environmental variables (config, etc.) that are not covered elsewhere in the bug tracking form.
- If you expect the reader to have any trouble reproducing the bug (special circumstances are required), be clear about them.

Bug Reporting: A Few More Fields

Resolution

The project manager owns this field. Common resolutions include:

- Pending: the bug is still being worked on.
- Fixed: the programmer says it's fixed. Now you should check it.
- Cannot reproduce: The programmer can't make the failure happen. Add details, reset the resolution to Pending, and notify the programmer.
- Deferred: It's a bug, but we'll fix it later.
- As Designed: The program works as it's supposed to.
- Need Info: The programmer needs more info from you. She has probably asked a question in the comments.
- Duplicate: This is just a repeat of another bug report (XREF it on this report.) Duplicates should not close until the duplicated bug closes.
- Withdrawn: The tester who reported this bug is withdrawing the report.

Comments

In many of the best databases, there are free-form comments fields that will take comments from anyone on the project.

- You will get comments (especially questions) from the project manager, the programmer and tech support. Other groups in your company might also be allowed to enter their comments.
- This field is especially valuable for recording progress and difficulties with difficult or politically charged bugs.
- Be cautious about your wording. Just like e-mail and usenet postings, it is easy to read a joke or a remark as a flame. Never flame.

Editing Bug Reports

Some groups have a second tester (usually a senior tester) review reported defects before they go to the programmer. The second tester:

- checks that critical information is present and intelligible
- checks whether she can reproduce the bug
- asks whether the report might be simplified, generalized or strengthened.

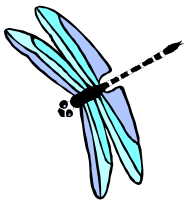
If there are problems, she takes the bug back to the original reporter.

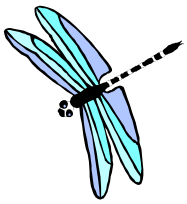
- If the reporter was outside the test group, she simply checks basic facts with him.
- If the reporter was a tester, she points out problems with an objective of furthering the tester's training.

This tester might review:

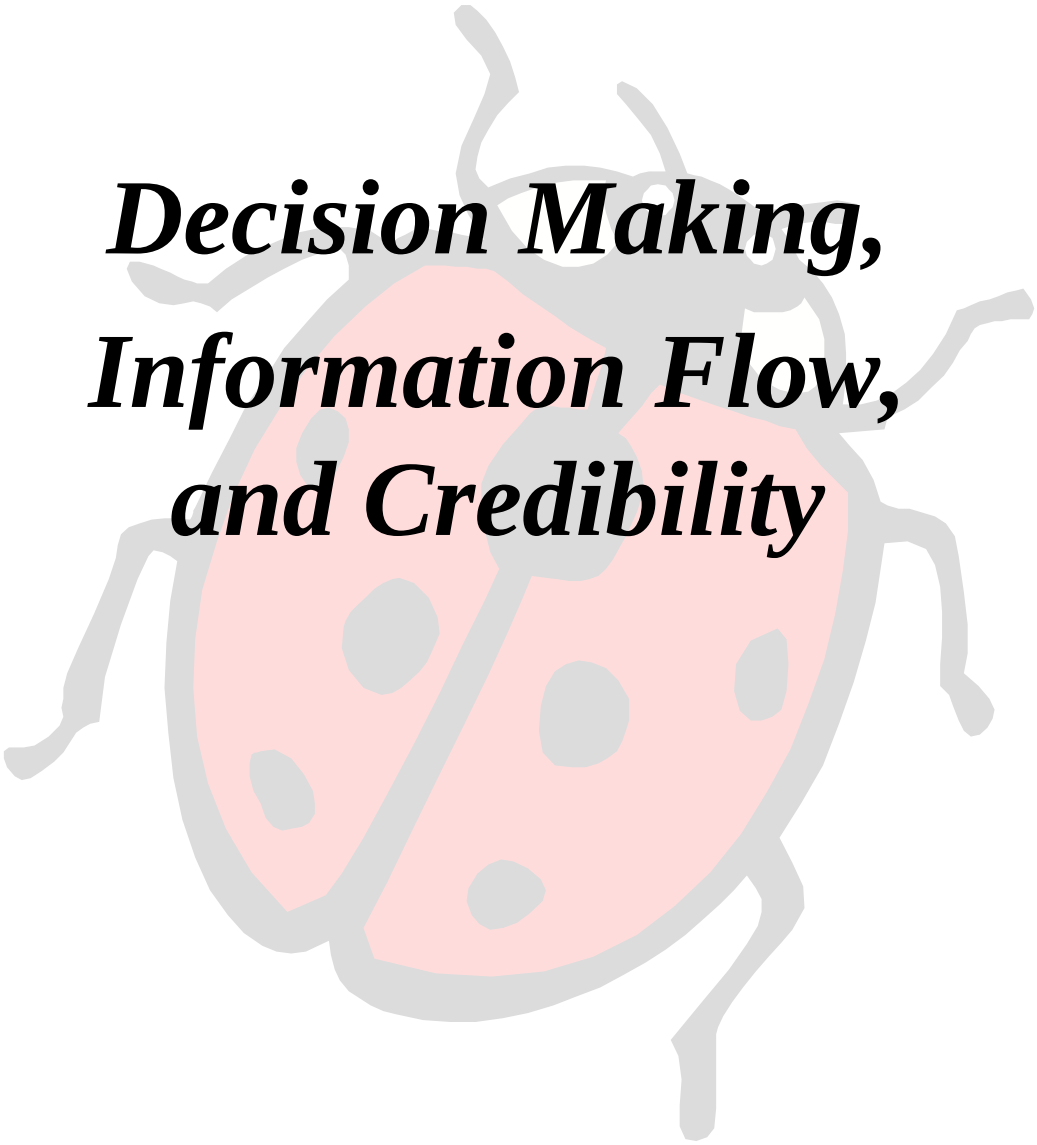
- all defects
- all defects in her area
- all of her buddy's defects.

In designing a system like this, beware of overburdening the reviewing testers. The reviewer will often go through a learning curve (learning about parts of the system or types of tests that she hasn't studied before). This takes time. Additionally, you have to decide whether the reviewer is doing an actual reproduction of the test or thinking about the plausibility and understandability of the report when she reads it.

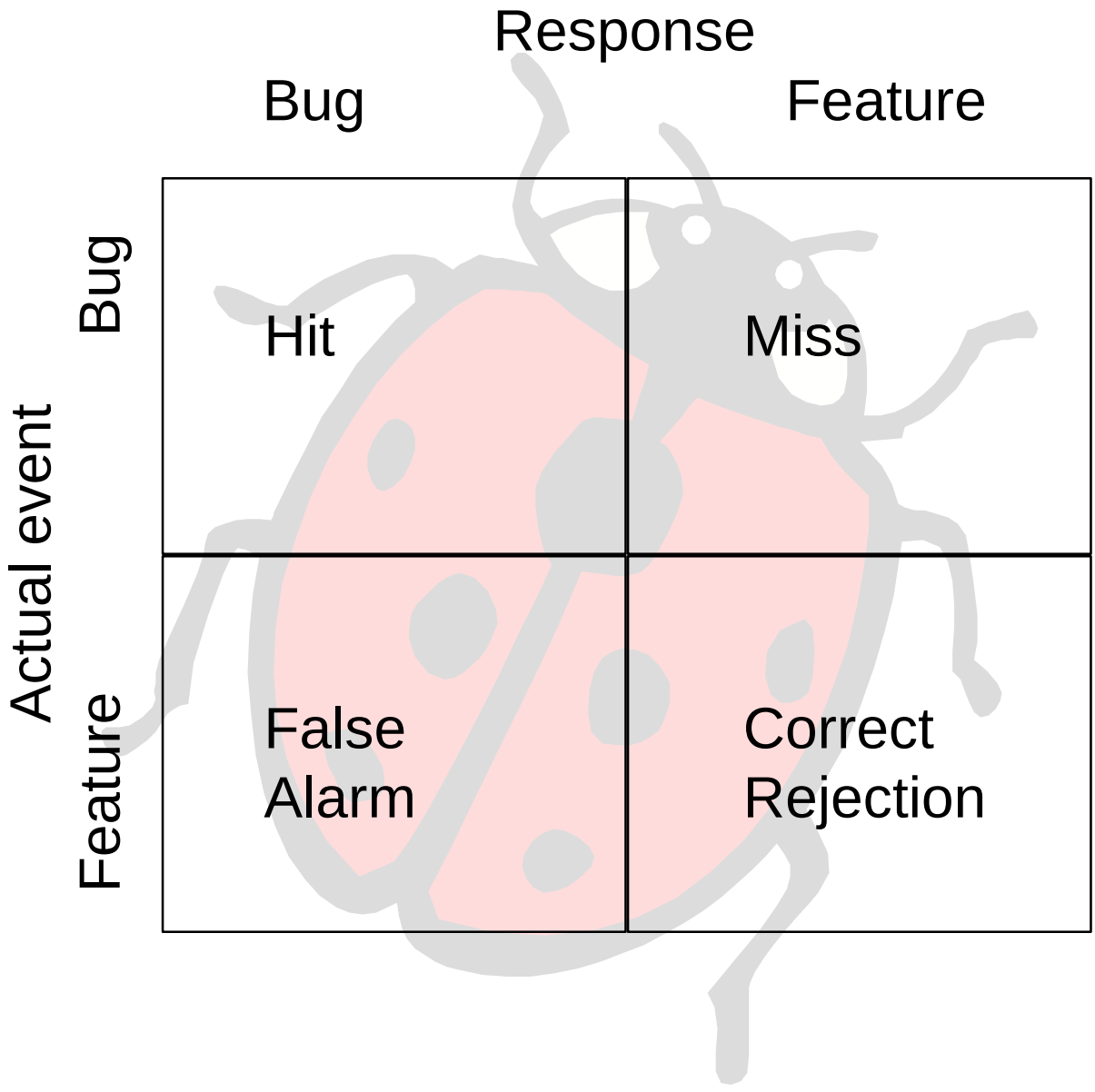




Decision Making, Information Flow, and Credibility



The Signal Detection & Recognition Problem



Refer to Testing Computer Software, pages 24, 116-118

Lessons From Signal Detection: We Make Decisions Under Uncertainty

When you try to decide whether an item belongs to one category or the other (bug or feature), your decision will be influenced by your expectations and your motivation.

- Can you cut down on the number of false alarms without increasing the number of misses?
- Can you increase the number of hits without increasing the number of false alarms?
- Pushing people to make fewer of one type of reporting error will inevitably result in an increase in another type of reporting error.
- Training, specs, etc. help, but the basic problem remains.

Lessons From Signal Detection: Decisions Are Subject To Bias

We make decisions under uncertainty.

Decisions are subject to bias, and much of this is unconscious.

The prime biasing variables are:

- ***perceived probability:***

  If you think that an event is unlikely, you will be ***substantially*** less likely (beyond the actual probability) to report it.

- ***perceived consequence of a decision:***

  What happens if you make a False Alarm? Is this worse than a Miss or less serious?

- ***perceived importance of the task:***

  The degree to which you care / don't care can affect your willingness to adopt a decision rule that you might otherwise be more skeptical about

Lessons From Signal Detection: Decisions Are Subject To Bias

Decisions are made by a series of people.

- **Bug reporting policies must consider the effects on the overall decision-making system, not just on the tester and first-level bug reader.**

Trace these factors through the decisions and decision-makers (next slides). For example, what happens to your reputation if you

- Report every bug, no matter how minor, in order to make sure that no bug is ever missed?
- Report only the serious problems (the "good bugs")?
- Fully analyze each bug?
- Only lightly analyze bugs?
- Insist that every bug get fixed?

Decisions Made During Bug Processing

Bug handling involves many decisions by different people, such as:

Tester:

- Should I report this bug?
- Should I report these similar bugs as one bug or many?
- Should I report this awkwardness in the user interface?
- Should I stop reporting bugs that look minor?
- How much time should I spend on analysis and styling of this report?

Your decisions will reflect on you. They will cumulatively have an effect on your credibility, because they reflect your judgment.

The comprehensibility of your reports and the extent and skill of your analysis will also have a substantial impact on your credibility.

Refer to Testing Computer Software, pages 90-97, 115-118

Decisions Made During Bug Processing-2

Bug handling involves many decisions by different people, such as:

Programmer:

- Should I fix this bug or defer it?

Project Manager:

- Should I approve the deferral of this bug?

Tester:

- Should I appeal the deferral of this bug?
- How much time should I spend analyzing this bug further?

Test Group Manager:

- Should I make an issue about this bug?
- Should I encourage my tester to
  - » investigate the bug further
  - » argue the bug further,
  - » or to quit worrying about this one,
  - » or should I just keep out of the discussion this time?

Refer to Testing Computer Software, pages 90-97, 115-118

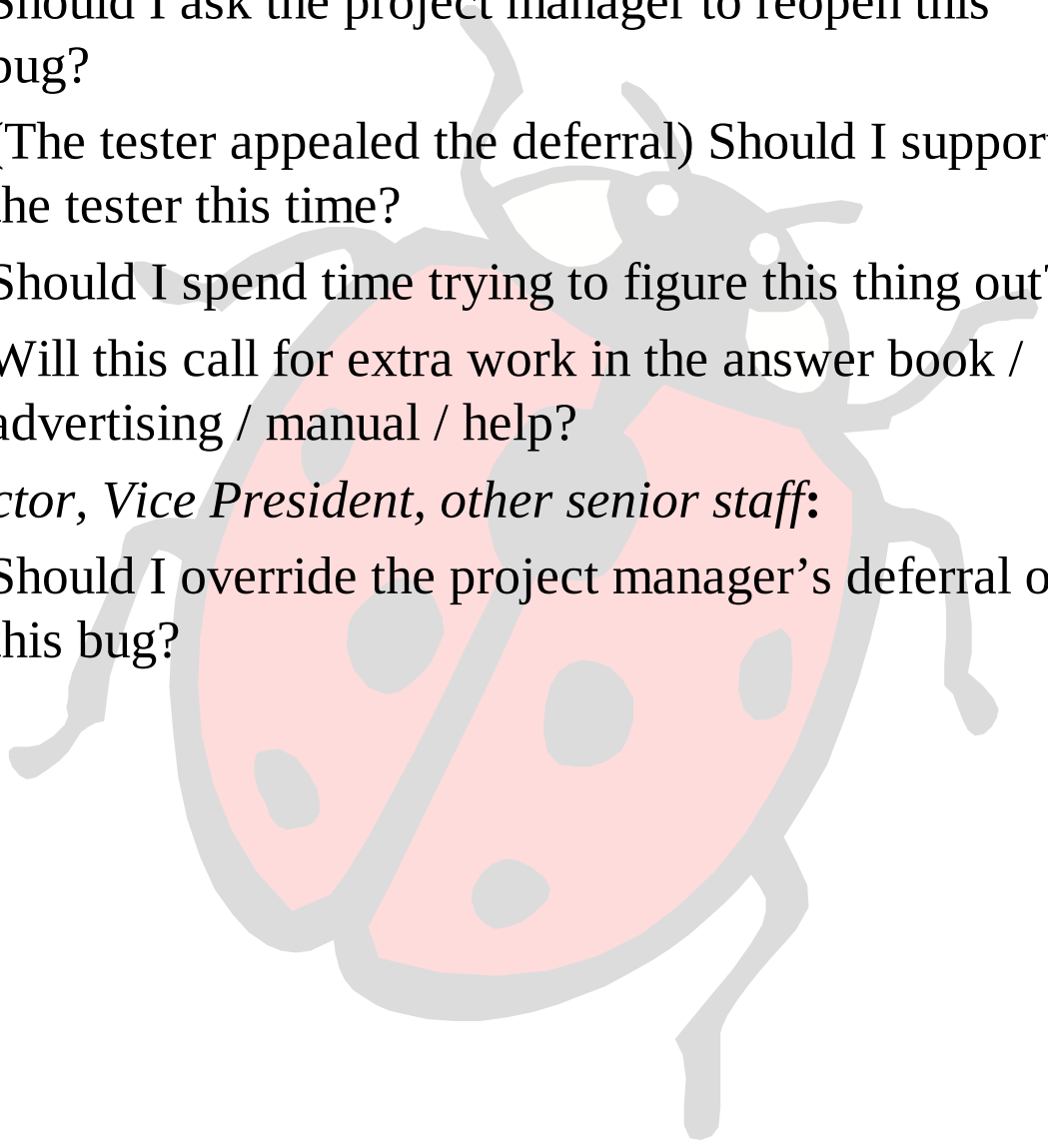
Decisions Made During Bug Processing - 3

Customer Service, Marketing, Documentation:

- Should I ask the project manager to reopen this bug?
- (The tester appealed the deferral) Should I support the tester this time?
- Should I spend time trying to figure this thing out?
- Will this call for extra work in the answer book / advertising / manual / help?

Director, Vice President, other senior staff:

- Should I override the project manager's deferral of this bug?



Decisions Made During Bug Processing - 4

Who else is in your decision loop?

Watch Out For Issues That Will Bias People Who Evaluate Bug Reports

These reduce the probability that the bug will be taken seriously and fixed.

- **Language critical of the programmer.**
- **Severity inflation.**
- **Pestering & refusing to ever take "No" for an answer.**
- **Tight schedule.**
- **Incomprehensibility, excessive detail, or apparent narrowness of the report.**
- **Weak reputation of the reporter.**

Watch Out For Issues That Will Bias People Who Evaluate Bug Reports

These increase the probability that the bug will be taken seriously and fixed.

- **Reliability requirements in this market.**
- **Ties to real-world applications.**
- **Report from customer/beta rather than from development.**
- **Strong reputation of the reporter.**
- **Weak reputation of the programmer.**
- **Poor quality/performance comparing to competitive product(s).**
- **News of litigation in the press.**

Clarify Expectations

One of the important tasks of a test manager is to clarify everyone's understanding of the use of the bug tracking database and to facilitate agreements that this approach is acceptable to the stakeholders.

- Track open issues / tasks or just bugs?
- Track documentation issues or just code?
- Track minor issues late in the schedule or not?
- Track issues outside of the published spec and requirements or not?
- How to deal with similarity?

Make the rules explicit.

Biasing People Who Report Bugs

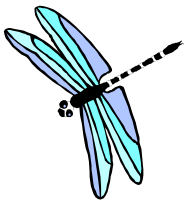
These help increase the probability that people will report bugs.

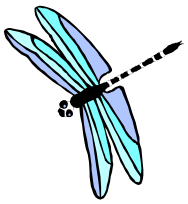
- Give results feedback to non-testers who report bugs.
- Encourage testers to report all anomalies.
- Adopt a formal system for challenging bug deferrals.
- Weigh schedule urgency consequences against an appraisal of quality costs. (Early in the schedule, people will report more bugs; later people will be more hesitant to report minor problems).
- Late in the schedule, set up a separate database for design issues (which will be evaluated for the start of the next release).

Biasing People Who Report Bugs

These will reduce the probability that bugs will be reported, either by discouraging reporters, by convincing them that their work is pointless or will be filtered out, or by creating incentives for other people to pressure people not to report bugs.

- Never use bug statistics for employee bonus or discipline.
- Never use bug statistics to embarrass people.
- Never filter reports that you disagree with.
- Never change an in-house bug reporter's language, or at least not without free permission. Add your comments as additional notes, not as replacement text.
- Monitor language in the reports that is critical of the programmer or the tester.
- Beware of accepting lowball estimates of bug probabilities.





STATUS REPORTS

are an effective tool for bug advocacy

Project Status

Bug Statistics are Only One Part of Status.

- Product status reporting is a long-term opportunity to communicate project information to a range of managers.
- Put the key issues that you want people to see on page 1 of the status report. Put the statistics later. People will flip a page or two to get to your statistics.

Project Status

My weekly status reports look like this:

A. Key Issues

- Deliveries needed
  - » What promises are still open
  - » What documents are still needed
  - » What functions are still uncoded
  - » What tools are not yet working
- Decisions needed
- **Bug fixes needed**
- Unexpected problems

B. Progress Against Plan

- Milestones
- Weekly goals
- Percent complete

C. Bug Numbers

D. List of Bugs Not Fixed

