The Test Manager at the Project Status Meeting

Brian Marick, Testing Foundations (marick@testing.com)
Steve Stukenborg, Pure Atria (stuke@pureatria.com)

Project management must decide when a product is ready to release to end users[SS1]. That decision is based on a mass of inevitably imperfect information, including an estimate of the product's bugginess. The testing team creates that estimate. The test manager reports it, together with an assessment of how accurate it is. If it's not accurate enough, because not enough testing has been done, the test manager must also report on what the testing team is doing to obtain better information. When will project management know enough to make a reasoned ship decision?

The test manager has two responsibilities: to report the right information, and to report the information right. Both are equally important. If you don't have the right information, you're not doing your job. If you don't report it so that it's accepted, valued, and acted upon, you might as well not be doing your job.

To address both topics, this paper is organized around the situation in which the test manager presents "the public face" of the testing team: the status meeting. It covers the information to be presented and the needs of the people involved (which govern how to present that information usefully).

1. Context

We assume a product developed in two distinct phases: feature development and product stabilization. We are specifically concerned with status meetings during stabilization, which is when the pressure is highest. During the stabilization phase, developers should do nothing but fix bugs. Ideally, they do not add or change features.

Testers do three types of testing during stabilization:

1? **Planned testing.** The tester is assigned some functional area or property of the product (such as the "printing subsystem" or "response to heavy network load"). At the beginning of the task, he knows the approach to take, what a "complete" set of tests would look like, and how much time is allocated. (This knowledge, like anything else in the project, is subject to revision, but it should be roughly correct.)

2? **Guerrilla testing.** This is testing that opportunistically seeks to find severe bugs wherever they may be. Guerrilla testing is much less planned than the previous kind. The extent of planning might be: "Jane, you're the most experienced tester, and you have a knack for finding bugs. From now until the end of the project, bash away at whatever seem to be the shakiest parts of the product." Guerrilla tests are usually not documented or preserved. See [Kaner93] for more information.

3? **Regression testing.** The tester reruns tests to see if one that used to pass now fails. If so, some change has broken the product.

In the first part of stabilization, planned tests dominate. As stabilization proceeds, more and more regression testing is done. At the end of the project, the testing effort shifts entirely to regression testing and guerrilla testing.

For more about the stabilization phase, see [Cusumano96] and [Kaner93]. For various software development models, see [McConnell96].

2. A Note on Numbers

Throughout the paper, we ask you to compare actual results to your plan. For example, we recommend that you track what proportion of their time your team spends rerunning tests vs. how much you expected they'd spend. A natural question is, "Well, then, what proportion should we expect?" We're not going to answer that question because the answers vary and, to be useful, require project-specific context. On the other hand, we think it's terrible

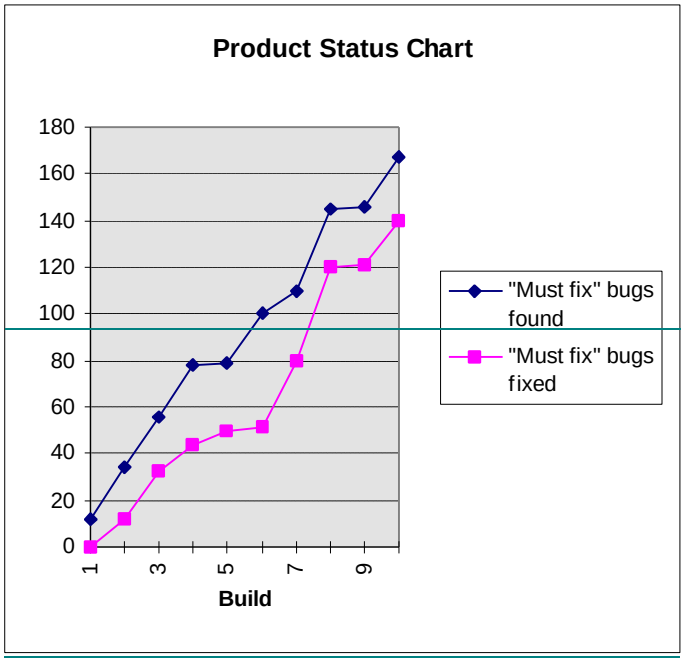
that test managers have nowhere to go to get those answers, other than to buttonhole their colleagues and swap war stories. For that reason, we will try to collect a variety of answers and rules of thumb from experienced test managers and put them on the web page http://www.stlabs.com/marick/root.htm.

3. Questions to Answer in the Status Meeting

During the project status meeting, you should be ready to answer certain questions, both explicit and implicit.

3.1 When will we be ready to ship?

During stabilization, successive builds are made, found not to be good enough, and replaced. It's a stressful time because, early in stabilization, it often seems as if the product will never be good enough. There's a lot of pressure to show forward progress. Much attention revolves around a graph like this:



Everyone wants to know when those lines will cross, meaning all must-fix bugs have been fixed.[1] Part of the answer comes from knowing how fast open bugs will be resolved. But another part of it comes from knowing how many new must-fix bugs can be expected. Our argument in this section is as follows:

1? Estimating this number is extremely difficult.
2? If you don't do it, someone else will, probably worse than you would. So you should do it.
3? Slight changes to the "intuitive way" of test planning and tracking can make the estimates better. So make those changes, as long as they don't get in the way of finding bugs.

To estimate this number, the natural tendency is to do some sort of curve-fitting (by using a ruler, a spreadsheet, or some more sophisticated tool). If you do, your extrapolations will likely be wildly wrong, because that curve is due to a changing mixture of completed tasks, tasks in progress, and tasks not yet begun, each with a different rate of

[1] At that point, all tests should have been designed and run at least once. The project shifts into the endgame, in which testers find must-fix bugs through regression tests and guerrilla testing, the programmers fix them, and builds are quickly turned over to the testers for another go-round.

finding bugs. Better than extrapolating from the summary graph is extrapolating from each testing task individually, using a spreadsheet like this:

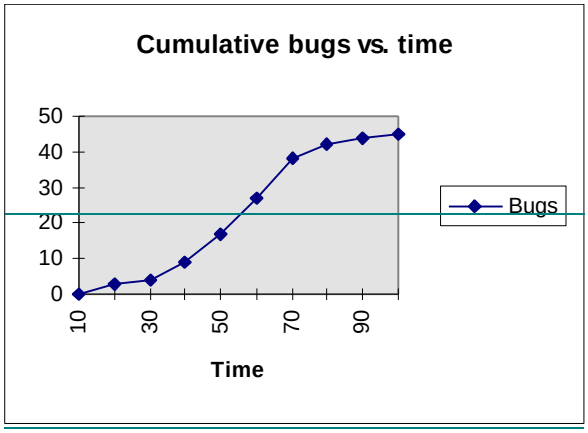
Task	% done	fatal bugs found	predicted
printing	40%	12	30
editing	60%	78	130
configuration	30%	3	10
scenario	50%	14	28
TOTAL		107	198

Important

What's being estimated with this spreadsheet is the total number of bugs *this testing effort will find before it's finished.* It is not the total number of bugs in the product. It is not the number of failures customers will see or report. Those would be better numbers to know, but this one is nevertheless useful.

Comparing apples and oranges

The spreadsheet above assumes that bugs will be found at roughly the same rate throughout a task. That's unlikely. A typical graph of cumulative bugs discovered vs. working time (which is more meaningful than calendar time) will look like this:



The reason for the curve is that different activities are lumped together in one task. Early in the task, the tester is learning an area of the product and designing tests. While test design does find bugs, the rate of bug discovery is lower than when the tests are run for the first time. So, when test execution begins, the curve turns up. As testing proceeds, an increasing amount of time will be spent in regression testing (especially checking bug fixes). Because rerunning a test is less likely to find a bug than running it for the first time, the curve begins to flatten out.

Your estimates will be more accurate if you track test design, first-time execution, and regression testing separately.

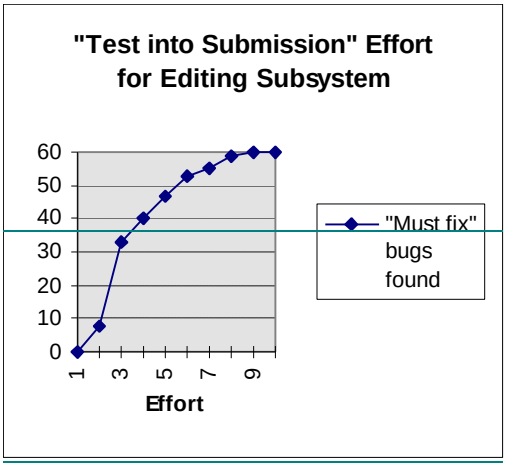
Notes:

1? We don't pretend that test execution doesn't have an element of design in it - while running tests, testers get ideas for new tests, which they then execute. But counting that "embedded design" as test execution should do no harm to what are already rough estimates. Major discoveries should make you schedule additional explicit tasks to be tracked separately.

2? The beginning of test execution will probably find more bugs than the end, because it's common for testers to run the most promising tests first. Even if tests were run in random order, some bugs are obvious enough that many of the tests will find them. While this "front-loading" of bugs might skew your estimates, you certainly don't want to improve them by intentionally finding bugs later!

3? You might find it most useful not to track retesting of bug fixes. Instead, simply assume that some percentage of fixed bugs will cause new bugs that will later be found by retesting.[2] Discover the percentage not by searching the literature (the numbers vary widely) but by reference to experience with your own project or other projects in your company.

4? In guerrilla testing, in which there's no formal design nor much rerunning of tests, the bug discovery rate nevertheless usually follows an S-shaped pattern, if not a more sawtooth pattern as the tester switches from opportunity to opportunity:



A linear estimate early in the effort would guess high. That's not necessarily bad - not because it gives you more breathing room, but because the harm of estimating high is less than the harm of estimating low. You should temper the estimate with some judgment about when the curve is likely to level off.

[2] They might be found by targeted retesting just after the bug is declared fixed, or they might be found by later whole-product regression tests that are run after a major build.

Reality Check

You may object that this approach seems hopelessly idealistic. Your project is chaotic enough, with code that changes so much, and so many bugs, and so much churn in the features, and... that there's no hope of having tasks that stay the same long enough for extrapolations to be meaningful. If that's the case - if your stabilization phase is anything but stable and you're really doing mostly guerrilla testing - there's no point in presenting wildly inaccurate estimates at the project status meeting. Concentrate on answering the other questions presented in this paper.

What about unstarted tasks?

Here's a different spreadsheet, one for a project in which configuration testing hasn't started yet:

Task	% done	fatal bugs found	predicted
printing	40%	12	30
editing	60%	78	130
configuration	0%	0	#DIV/0!
scenario	50%	14	28
TOTAL		104	#DIV/0!

Not useful

If you haven't started a task, you have no data from which to extrapolate. But you have to, because the project manager needs estimates in order to run the project. Here are some solutions:

- Don't get into this situation. By the time the stabilization phase is well enough underway that this graph becomes the focus of attention, you should have made some progress on all testing tasks. That almost certainly means some context switching. For example, the tester responsible for the scenario testing task will have to stop part-way through and do some configuration testing. That's annoying, disruptive, and reduces efficiency - but only if you take a tester-centric view of the universe, one where getting testing done is the primary goal. Your team's primary goal is discovering useful information and making it available as early as possible. The project manager needs to know if there are configuration problems before the last minute.

- Extrapolate using numbers for testing that's already underway. For example, both "test printing" and "test editing" are functional tests of discrete subsystems. If the printing subsystem is half the size of the editing subsystem, and half as much testing is planned, you might reasonable predict half as many fatal bugs will be discovered. There are, of course, many variables that might confound that estimate: the programmer who wrote the printing code may be a much better programmer, the size metric you used (be it lines of code, number of branches, function points, programmer hours spent) may not capture the relative bugginess of the two subsystems, and so on. However, it's a better estimate than simply guessing, and it will improve rapidly as real testing begins.

- Extrapolate using numbers from past projects. We recommend that anyone doing any sort of data mining read up on the statistical subfield of exploratory data analysis. Two entertaining and influential early books are [Tukey77] and [Mosteller77].

Reality check: you will certainly be wrong

Even if you are careful with your extrapolation, you should realize that the number calculated is certainly wrong. Attaching too much certainty to the numbers - and especially striving for meaningless precision - will lead to madness. Concentrate on testing, and hope that the errors in your estimates roughly cancel each other out.

Ideally, you'd like to give your estimates in the form of intervals, like this:

Week 1	"We predict 122 to 242 bugs. Best estimate is 182."
Week 2	"We predict 142 to 242 bugs. Best estimate is 192."

Week 3	"We predict 173 to 233 bugs. Best estimate is 203."
Week 4	"We predict 188 to 228 bugs. Best estimate is 208."
Week 5	"We predict 205 to 215 bugs. Best estimate is 210."

Unfortunately, we expect your data will be unstable enough that your calculated "confidence intervals" will be so wide as to be pointless. We're also unaware of statistical techniques that apply to this particular problem. If you wish to explore related notions, see the large body of literature on software reliability and software reliability growth ([Musa87][Lyu96]). Your best bet might be using historical project data. If your company keeps good records of past projects, and has a reasonably consistent process, you may be able to use errors in historical estimates to make predictions for this project.

We think that, rather than straining for numbers far more exact than anything else in the project, you should concentrate on understanding and explaining what caused any recent changes in the estimates. Here, your goal is confidence - in you and your team - not confidence intervals. You want to say things like:

> "Last week's prediction was roughly 69 more must-fix bugs. However, Dawn noticed some oddities as she continued the testing of the modules from our outsource partners, NightFly Technologies. She beefed up her test plan with some targeted use case tests. That led to last week's jump in bugs found, and it raised our prediction to roughly 95 bugs."

The description behind the numbers is *much* more useful than the numbers, because it helps the project manager make contingency plans.

Final notes on estimation

1? Toward the end of the stabilization phase, your estimates will become useless. That's the point at which your team is concentrating on regression tests for the last few bug fixes, together with guerrilla testing that tries to squeeze out a few more must-fix bugs. This period is inherently unpredictable - a single lingering bug could take a month to fix. It's the worst part of the project. You will still have provided a valuable service by helping to predict when the hellish period starts.

2? [DeMarco82] has useful things to say about estimation. [Bach94] describes a bug tracking system initially used to produce estimates of the sort described here. See also the description of Bach's work in [Keuffel94], which says, "Bach now believes too many independent variables exist to make useful predictions of ship dates by simply leveraging off the critical-bug database." [Keuffel95] continues the description with some interesting deductions you can make from the shape of a project-wide graph.

3? Make sure that the graph of found vs. fixed "must fix" bugs doesn't dominate attention to the exclusion of other shipping criteria. Other criteria that are sometimes used include trends in number of bugs found, active, fixed, and verified; number of lower-severity unfixed bugs; changes in bug severity distribution; and amount of recent change to the source code ([Cusumano95], [Kaner93], [Rothman96]). And the product shouldn't ship until testing is finished, including regression testing and a last-chance guerrilla test of the final build.

3.2 Which bugs should be fixed?

A discussion of individual bugs might happen in the project meeting or in a separate bug classification meeting. (This might be a discussion of all bugs or, more productively, a discussion of only controversial bugs.) You are likely to be the person who prints and distributes the summaries and detailed listings of new and unresolved bugs. It's preferable if the project manager leads the meeting, not you. First, he or she has the ultimate responsibility. Second, you will be an active participant, and it's difficult to do a good job as both participant and meeting leader.

Here are some key points:

1? Developers have a psychological interest in assigning bugs a low priority or deferring them. However, you may well tend to over-emphasize the severity of a bug, which is just as harmful. The best advocates for fixing a bug are the people who will have to live with its consequences. That's not you. It's more likely to be customer service people and marketing people. They should be invited to bug classification meetings. Your role as advocate is to make sure that the bug report is understood correctly. If the bug report is being misinterpreted (for example, as being an unlikely special case instead of a symptom of a more general problem), correct the misinterpretation.[3]

2? The key issue in such a meeting is determining whether the risk of fixing a bug exceeds the risk of shipping it with the product. When helping assess risk, your job is to provide two bits of informed opinion:

- What are the testing implications of fixing the bug (whether it's fixed alone or batched with other bugs in the same subsystem)? How much will adequately testing the fix cost? Do you have the resources to do it? Is it a matter of rerunning regression tests? (What's the backlog?) Or will new tests have to be written?
- What's the rate of regressions? How often do bug fixes fail? (If the rate of regressions is high, especially in the subsystem in question, and that subsystem has light test coverage, the risk of a bugfix is high.)

3? Higher management may be tracking open bug counts. That can lead to unproductive pressure to make the numbers look good at the expense of fixing fewer but more important bugs. (See [Kaner93], chapter six.) As a responsible project member, you should be alert to decisions that provide only internal benefit, not benefit to the customer.

We urge you to read the discussions of bugs and bug reporting in [Bach94] and [Kaner93].

3.3 Where are the danger areas in the project?

As part of the estimation process, you will become aware of "hot spots" in the product: areas where there are unusually many bugs. In the charts above, testing editing is finding far more bugs than any other testing task.

The project manager is sometimes the last to know about hot spots, since developers are incorrigibly optimistic. They tend not to admit (to themselves) that a particular subsystem is out of control until too late. It's your job as a test manager to report on hot spots at the status meeting. You need to do this with care, lest your team be viewed as the enemy by development. Here are some tips.

- **Make sure it really is a hot spot.**

  The heat of a hot spot is relative to the size and complexity of the area under test. If the editing subsystem is four times as complex as the printing subsystem, four times as many bugs is not surprising.[4] You should have complexity information available, because you should have used it in test planning.

  Historical information will help to keep hot spots in perspective. What's the bug rate for previous projects (per line of code, function point, or some other measure of size)? What parts of this project are out of line with the historical averages? [5]

[3] There are cases in which you must live with the consequences of a bug. It may be blocking further planned testing. It may break existing test automation and thus increase the cost of regression testing. In such cases, you should make the consequences known.

[4] You scheduled four times as much testing for the editing subsystem because it's four times as big.

[5] You quite likely do not have this information now. But you could start collecting it now.

- **Report on product, not people.**

  With perhaps a few exceptions, such as configuration testing, the hot spots you find will be associated with a particular person or team. Don't criticize them. Maybe they've done a shoddy job, but don't assume you have enough information to make that judgement. Even if you do, it's counterproductive at this point.

  Not only must you not criticize the developers, it's important that they not think you're criticizing them. Your manner should be calm, dispassionate, and professional. Use "we" instead of "you". You should take care to say something good along with the bad news:

  > "That last overhaul of the account management module has really streamlined the user interface. Paul, the tester, says that it takes half as many mouse moves to get through a test as before. Of course, that means that users will see the same speedup. Unfortunately, his tests have found a big jump in the number of bugs, to the point where we're well above average for modules in this product."[6]

- **Warn of hot spots in advance.**

  Do not announce a hot spot without prior warning. You know you've damaged your usefulness when the project manager explodes, "What do you mean, the networking subsystem is in crisis! Why haven't I heard about this before?!"

  One reason for no prior warning is that there's no data. Some testing task hadn't started until the past week. When it did start, the tester involved found so many bugs she had to down tools right away. Avoid the issue by doing at least a little testing on all tasks as soon as possible. You can defuse the issue by reporting, at each status meeting, which important tasks haven't started, along with a prediction of when each will start.

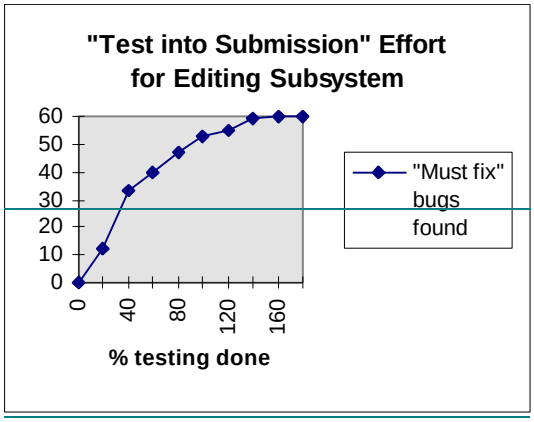
  Another reason for a sudden announcement is that a testing task suddenly crosses the threshold where you feel you should raise the alarm. Avoid this by announcing which tasks you're watching more carefully, though they don't yet merit the designation "hot spot".

- **Track hot spots carefully.**

  The response to a hot spot should be some development effort (redesign or rework). When that's complete, report on the success: "We reran all the regression tests on the new build, with unusually few regressions, which personally gives me a warm and fuzzy feeling. We've started executing new tests on the module, and things look good so far - many fewer bugs than in the last version. I think we're out of the woods on this one."

  An alternative to rework is to "debug the code into working". In that case, you need to adjust your plans. Whereas before you scheduled a fixed amount of testing, you now have an open-ended testing task. You might increase the number of designed tests; you will certainly devote more time to guerrilla testing. You need to continue testing until you stop discovering a disproportionately large number of bugs. You will look for a graph like this:

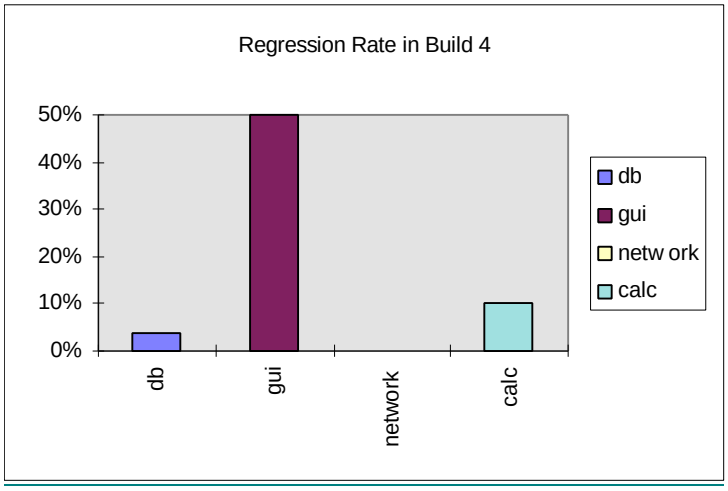
[6] Yes, this is entirely transparent. Everyone knows about the "pair every criticism with a compliment" technique. Doesn't matter. They'll still be pleased that you have respect for their feelings.



After nearly twice as much testing as planned, you seem to have exhausted the bugs in the code.

Another type of hot spot

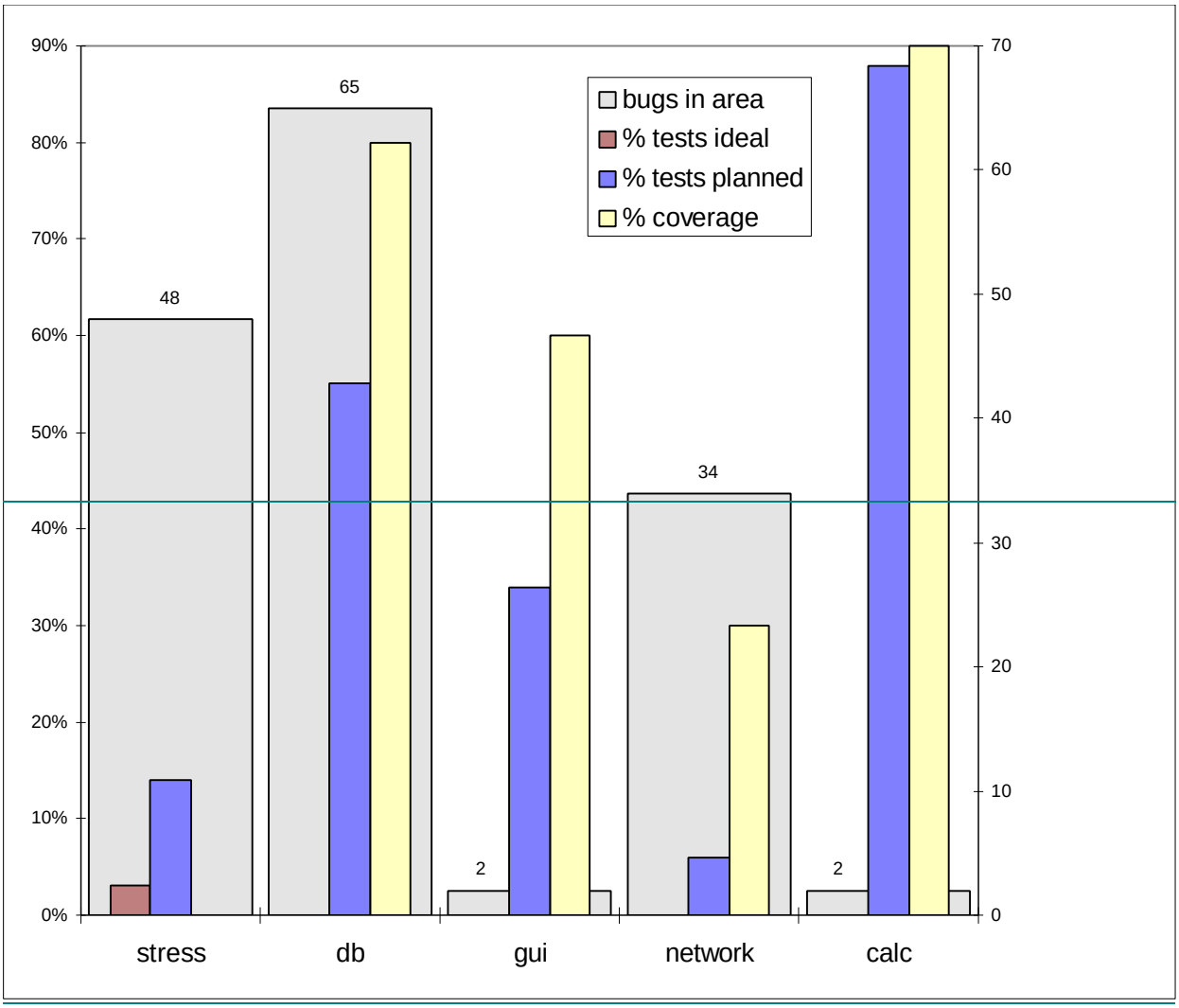
In addition to total bugs, track "churn": how many regression tests fail. If a disproportionately high number of bug fixes or other changes break what used to work, there's a problem. The subsystem is likely fragile and deserves rework or a greater testing effort. Here's a chart that might cause alarm:



3.4 Are you working smart?

Any buggy area announced late in the project is going to be stressful. It will likely mean some upheaval, some shifting of development priorities. As test manager, you must also shift your priorities, both because you now know more about risky areas and also because you must be seen as doing more than smugly announcing a crisis and letting someone else deal with it. It's important to both be helpful and also be seen to be helpful.

Display graphs to show that the testing team is flexibly adapting what you do to what you discover. One such graph follows. It breaks testing down by task. It plots effectiveness (bugs found) behind measures of effort to date. That way, mismatches between effort and result are visually striking. The graph uses two measures of effort: time spent and *coverage* achieved. Code coverage measures how thoroughly tests exercise the code. That correlates roughly with thoroughness of testing. The most common type of code coverage measures which lines of code have been executed. For other types of coverage, see [Kaner96] and [Marick95]. For a discussion of how coverage can be misused, see [Marick97].



Here's what you might say in the project status meeting. (Notice that the individual testing activities tracked separately for bug count estimation are here grouped into larger tasks.)

"The stress tests and db tests are paying off in terms of bugs found. The stress tests are only 14% complete, so you can expect a lot of stress-type bugs in the next few weeks. You'll notice that we early in the project cut back on the number of stress tests - although we're 14% complete against plan, that's only *three percent* of what we originally hoped to do. If stress testing keeps finding bugs, we'll up the amount of stress testing planned.

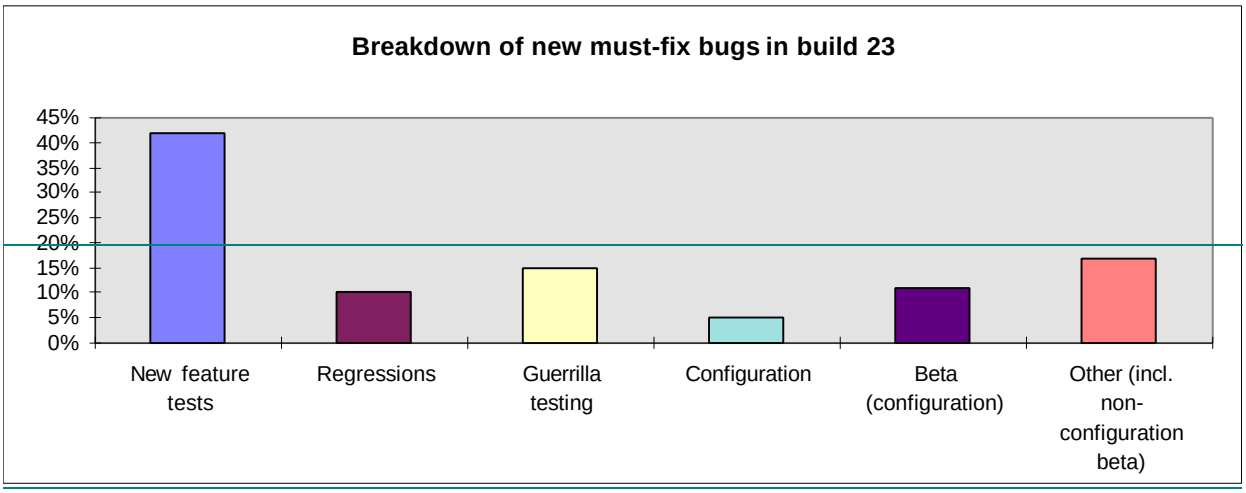
"The db tests are 55% complete against plan, and they've got 80% coverage. Given the type of testing we're doing, we might see the bug finding rate start leveling off soon. We may not do all the testing planned.

"The GUI tests have been a washout. We're not finding bugs. The code seems solid. We're stopping that testing now and shifting the tester onto network testing, where we're finding tons of bugs. In the network testing, you'll note that the coverage is high in comparison to the number of tests written. That's because we've finished the "normal use tests". The error handling tests, which we couldn't start until the network simulator was ready, will add less coverage, but we expect them to be a good source of bugs.

"The calc tests were a botch. We're nearly done, without much to show for it. My fault - I let the schedule get away from me. We're stopping testing right away."

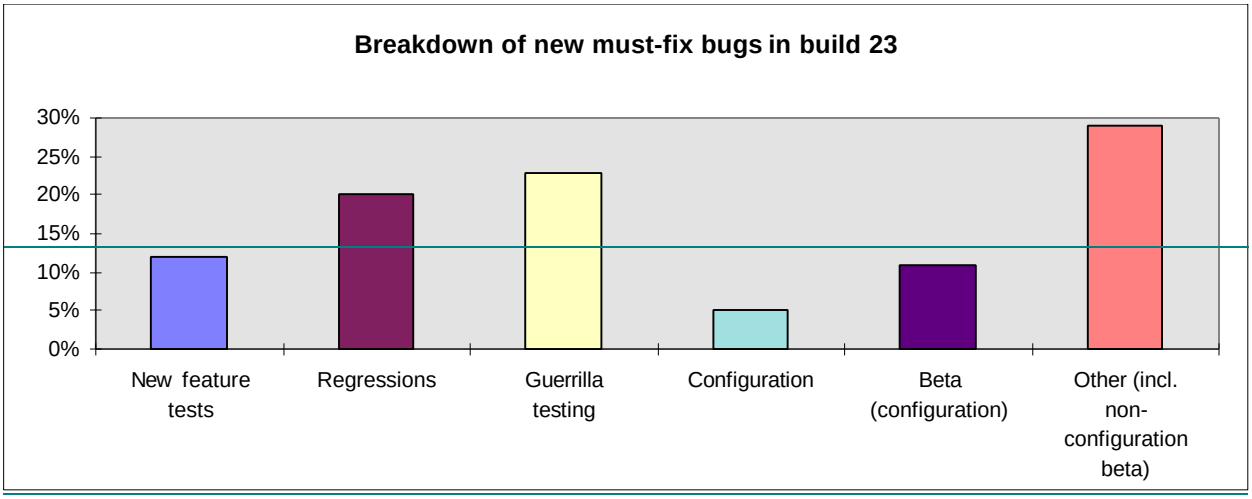
Comparing yourself to others

It's a common developer prejudice that testers write a lot of tests, but it's wasted motion because they're not finding the important bugs. First, the test manager needs to check whether the prejudice is justified. (All too often, it is.) Second, if it's not, reassure the project team. Doing both things starts with a graph like this, which shows what percentage of bugs are being found by different types of planned testing vs. those found by chance.



This shows a fairly reasonable breakdown. Most new bugs are found by the testing team: newly written feature tests, regression tests run again, guerrilla tests, and planned configuration tests. 10% of new bugs reported are due to regressions, which is not outlandish. Beta customers are reporting configuration bugs at a good clip, which is what you expect from them. (You might want to consider doing more in-house configuration testing if the total number of configuration bugs is unusually high.) There are a reasonable number of bugs from other sources (beta bugs that are not configuration problems, bugs found by developers and other in-house use, and so on).

This graph is more alarming:



Too many bugs are being found by "other", not enough by the testing team. They're being discovered by chance. Can the testing effort change to detect them more reliably? The fact that guerrilla testing is doing so much better than the new feature tests is also evidence that something's wrong with the planned testing effort. Perhaps the problem is that the project is "churning": there are too many regressions. Maybe almost everyone on the testing team is spending so much time running regression tests that they have little time to execute new ones (except for a few very productive guerrilla testers).

When many bugs have been found, the testing team can become a bottleneck. Report clearly when regression testing load exceeds the plan. Such reports, made early and forthrightly, leave the impression that slips in the testing schedule are a result of pitching in to help the project. Done late, you may get the blame for the final slips, slips that happen as developers wait for you to finish up your testing.

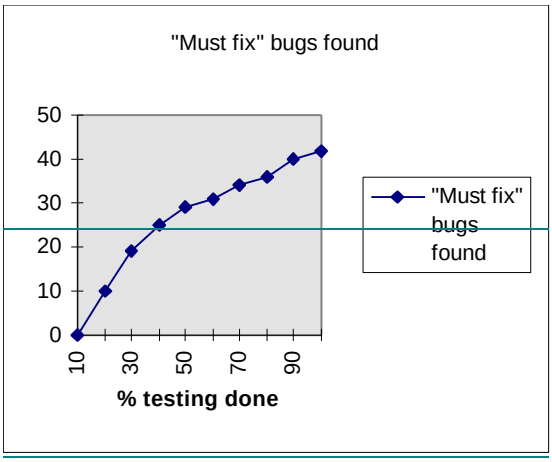
When reporting on your schedule, remind everyone that the earliest possible ship date is the time required to complete all planned tests, plus the time required to rerun all repeatable tests (both manual and automated) on the final build, and time for a series of guerrilla tests on that build.

3.5 What does your work mean?

We've seen how you can predict the number of bugs testing will find and use graphs to reallocate the testing effort appropriately. But what does all this effort mean?

Recall that the project manager wants to reduce uncertainty and risk. He worries that this product will be an embarrassment or worse after it ships. Knowing how many bugs testing will find is useful, because it lets him predict whether the product can ship on time, but what he *really* wants to know is how many bugs testing *won't* find.

Let's assume that you graph the bugs found by new (not rerun) designed tests for some particular testing task. If that task is consistent throughout (you don't apply different testing techniques during different parts of the task, or save the trickiest tests for the end), you'd expect a graph that looks something like this:



You'd expect something of a flurry of new bugs at first. Many of those are the bugs that are easy to find, so that many different tests would hit them. After those have been disposed of, there's a more-or-less steady stream of new bugs. (In practice, the curve might tail down more sharply, since there's a tendency to concentrate on the most productive tests first, in order to get bugs to the developers quickly.)

This graph doesn't really let you predict what happens in the field. You can't simply extrapolate the curve. If you did a perfect job of test design, you've tried all the bug-revealing cases, and all bugs have been found. If your test cases are very unrepresentative of typical customer use, you might have missed many of the bugs customers will find, in which case there will be a surge of new bug reports when the product is released (not unusual).

So what can you do? You have to extrapolate from previous projects.

- If you discovered 70% of the configuration bugs during test in release 3.0, and if you continue to perform configuration testing roughly the same way, the safest bet is that you'll again miss 30% of the bugs.
- If you consistently apply the same set of testing techniques, and you size the testing effort against the product in a consistent way, you should hope that you'll find the same proportion of bugs from effort to effort. In practice, of course, the variability among testers "doing the same thing" is as large as it is for programmers (that is to say, enormous). You can reduce variability by having testers review each other's test designs, but you won't eliminate it. That's OK. Saying "based on the past three projects, we predict between 50 and 120 more must-

fix bugs will be found by customers" beats only being able to say "We tested hard, and we hope we didn't miss too much."

When reporting extrapolations, don't give raw numbers. Say, "we predict the reliability of this product will be typical [or high, or low] for this company." If possible, make comparisons to specific products: "This product did about average on the planned tests, but remarkably few bugs were found in guerrilla testing or regression testing. The last product with that pattern was FuzzBuster 2.0, which did quite well in the field. We predict the same for this one."

The accuracy of your predictions depends on a consistent process. Consistent processes are the subject of industrial quality control, especially the work of Shewhart, Deming, Juran, and others. We in software can learn much from them, so long as we're careful not to draw unthinking analogies. [Deming82] is often cited, but the style is somewhat telegraphic. [Aguayo91] has been recommended to one of us (Marick), who first learned about statistical quality control from a draft of [Devor91]. The Software Engineering Institute's Capability Maturity Model can be thought of as an adaptation of modern industrial quality control to software, though the coverage of testing is sketchy. See [Humphry89] or [Curtis95].

4. Summary of the paper

As the frequent bearer of bad news, the test manager has a difficult job. The job is easier if he or she appears competent, knowledgeable, and proactive. It's a matter of having the relevant information, presenting it well, being able to answer reasonable questions. A goal should be to walk out of the status meeting knowing that the testing team and its work have been represented well.

As the test manager, you are the keeper of data that can help you understand trends and special occurrences in the project, provided you avoid the two "data traps": having unjustified faith in numbers, and rejecting numbers completely because they're imperfect. Take a balanced view and use data as a springboard to understanding.

5. Acknowledgements

Keith Davis made helpful comments on an earlier draft of this paper.

6. References

[Aguayo91]
Rafael Aguayo, *Dr. Deming*, Fireside / Simon and Schuster, 1991.

[Bach94]
James Bach, "Process Evolution in a Mad World," in *Proceedings of the Seventh International Quality Week*, (Software Research, San Francisco, CA), 1994.

[Curtis95]
Mark C. Paulk, Charles V. Weber, and Bill Curtis (ed), *The Capability Maturity Model: Guidelines for Improving the Software Process*, Addison-Wesley, 1995.

[Cusumano95]
M. Cusumano and R. Selby, *Microsoft Secrets*, Free Press, 1995.

[DeMarco82]
Tom DeMarco, *Controlling Software Projects: Management, Measurement, and Estimation*, Prentice Hall, 1982.

[Deming82]
J. Edwards Deming, *Out of the Crisis*, MIT Center for Advanced Engineering Study, 1982.

[Devor92]
Richard E. Devor, Tsong-How Chang, and John W. Sutherland, *Statistical Quality Design and Control: Contemporary Concepts and Methods*, MacMillan, 1992.

[Humphrey89]
Watts Humphrey, *Managing the Software Process*, Addison-Wesley, 1989.

[Kaner93]
C. Kaner, J. Falk, and H.Q. Nguyen, *Testing Computer Software (2/e)*, Van Nostrand Reinhold, 1993.

[Kaner96]
Cem Kaner, "Software Negligence & Testing Coverage," in *Proceedings of STAR 96*, (Software Quality Engineering, Jacksonville, FL), 1996. (http://www.kaner.com/coverage.htm)

[Keuffel94]
Warren Keuffel, "James Bach: Making Metrics Fly at Borland," *Software Development*, December 1994.

[Keuffel94]
Warren Keuffel, "Further Metrics Flights at Borland," *Software Development*, January 1995.

[Lyu96]
Michael R. Lyu (ed.), *Handbook of Software Reliability Engineering*, McGraw-Hill, 1996.

[Marick95]
Brian Marick, *The Craft of Software Testing*, Prentice Hall, 1995.

[Marick97]
Brian Marick, "Classic Testing Mistakes," in *Proceedings of STAR 97*, (Software Quality Engineering, Jacksonville, FL), 1997. (http://www.stlabs.com/~marick/root.htm)

[McConnell96]
Steve McConnell, *Rapid Development*, Microsoft Press, 1996.

[Mosteller77]
Frederick Mosteller and John W. Tukey, *Data Analysis and Regression*, Addison-Wesley, 1977.

[Musa87]
J. Musa, A. Iannino, and K. Okumoto, *Software Reliability : Measurement, Prediction, Application*, McGraw-Hill, 1987.

[Rothman96]
Johanna Rothman, "Measurements to Reduce Risk in Product Ship Decisions," in *Proceedings of the Ninth International Quality Week*, (Software Research, San Francisco, CA), 1996. (http://world.std.com/~jr/Papers/QW96.html)

[Tukey77]
John W. Tukey, *Exploratory Data Analysis*, Addison-Wesley, 1977.