CEM KANER, J.D., Ph.D.
Professor
Department of Computer Sciences
Florida Institute of Technology
150 West University Blvd., Melbourne, FL 32901
kaner@kaner.com www.kaner.com

Measurement Issues and Software Testing[1]

March, 2001

Note: This is a draft[2] of a chapter on testing and software measurement that will appear in the third edition of Testing Computer Software (by Cem Kaner, James Bach, Hung Quoc Nguyen, Jack Falk, Bob Johnson, and Brian Lawrence).

This particular draft was prepared for the QUEST conference in Orlando, FL.

We will extend the reference list substantially in the next draft. We expect this chapter to be controversial, but we very much want it to be fair and accurate. Please send comments (including flames, if necessary) to Cem Kaner at kaner@kaner.com.

---

[1] Much of the material in this chapter comes from participants of *Software Test Managers Roundtable* (STMR) and the *Los Altos Workshop on Software Testing* (LAWST).

–STMR 1 (October 3, November 1, 1999) focused on the question, ***How to deal with too many projects and not enough staff?*** Participants: Jim Bampos, Sue Bartlett, Jennifer Brock, David Gelperin, Payson Hall, George Hamblen, Mark Harding, Elisabeth Hendrickson, Kathy Iberle, Herb Isenberg, Jim Kandler, Cem Kaner, Brian Lawrence, Fran McKain, Steve Tolman and Jim Williams.

–STMR 2 (April 30, May 1, 2000) focused on the topic, ***Measuring the extent of testing***. Participants: James Bach, Jim Bampos, Bernie Berger, Jennifer Brock, Dorothy Graham, George Hamblen, Kathy Iberle, Jim Kandler, Cem Kaner, Brian Lawrence, Fran McKain, and Steve Tolman.

–LAWST 8 (December 4-5, 1999) focused on ***Measurement***. Participants: Chris Agruss, James Bach, Jaya Carl, Rochelle Grober, Payson Hall, Elisabeth Hendrickson, Doug Hoffman, III, Bob Johnson, Mark Johnson, Cem Kaner, Brian Lawrence, Brian Marick, Hung Nguyen, Bret Pettichord, Melora Svoboda, and Scott Vernon.

Portions or drafts of this chapter were published at the Pacific Northwest Software Quality Conference (Portland, 1999, 2000), the International Software Quality Week conference (San Francisco, 2000), and the XXX Segue QUEST International User Conference (Orlando, 2001). We also thank the many participants at the 1999 Consultants Camp (Crested Butte, CO) for their early review of this material, and Barbara Kitchenham for detailed comments on a later draft.

[2] In drafts, I use XXX to flag something that must be updated later. I hope that these flags are not too distracting.

Overview

Measurement is the assignment of numbers to objects or events according to a rule derived from a model or theory.

A theory of measurement must take into account at least 10 factors:

- The attribute to be measured
    - The appropriate scale for the attribute
    - The variation of the attribute
- The instrument that measures the attribute
    - The scale of the instrument
    - The variation of measurements made with this instrument
- The relationship between the attribute and the instrument
- The likely side effects of using this instrument to measure this attribute
- The purpose of taking this measurement
- The scope of this measurement

This chapter considers several measures in widespread use in software testing, such as:

- Use of bug counts to measure the effectiveness or efficiency of a tester
- Use of code-structure-based formulas to measure the complexity of a computer program.

Similar approaches have been laid out by other authors on software measurement (Kitchenham, Pfleeger, & Fenton, 1995; Zuse, 1997) and in other fields, such as physics (*see, e.g.* Sydenham, Hancock & Thorn, 1989) and psychometrics (*see, e.g.* Allen & Yen, 1979).

Our point of departure from the traditional *software* measurement literature is in the extent to which we ask (a) whether commonly used measures are valid, (b) how we can tell, and (c) what kinds of side effects are we likely to encounter if we use a measure? Even though many authors mention these issues, we don't think they've been explored in enough depth or that enough people are thoughtfully enough facing the risks associated with using a poor software measure (Austin, 1996; Hoffman, 2000).

After laying the groundwork, the chapter focuses on a typical test management problem--*How can we measure the extent of the testing done on a product*? We won't come up with *The Right Answer*, but we will confront a fact--management will demand measures of the extent of testing, and your testing group will have to provide the reports to management that they demand. We look at a wide range of measures that have been proposed, and at the structure of some status reports that have been used by successful test managers.

Measurement is difficult. It takes years of time, thought, modeling, and data collection to develop valid, useful measures. To put our situation (in Computer Science) in perspective, we've added an appendix to this chapter that describes the history of perceptual measurement (over nearly 150 years of work) in terms of the measurement factors listed above. This psychological work is of interest not just because it shows the gradual development of measurement theory but also because so many software-related measurements involve psychological or subjective components. To a large degree, these human factors are at the root of the complexity of measurement of what initially appear to be software-related, not human-related variables.

Introduction

Many people want to measure variables related to software testing. This is difficult Here are three simple examples that carry some publicity (or notoriety) in the field, and that illustrate the problems:

- ***Software Complexity:*** How complex (hard to understand, hard to maintain, etc.) is a given piece of software? How hard will it be to test? ***Is McCabe's "metric" a measure of software complexity?***
- ***Tester Goodness:*** How "good" is a given tester? If "good" is too vague, substitute "productive" or "effective" or "valuable" (or some similar concept of your choice). ***Is bug count a measure of the goodness of testers? Of the overall effectiveness or thoroughness of the testing effort? Of the degree to which the software is ship ready?***
- ***Extent of Testing:*** After testing a program for a few months, how much testing has been accomplished? ***Is code coverage a measure of the extent of testing?***

Claims have been made for each of these "measures" but how would we know if they were any good?

One of the core problems of measurement in computing is that many of the things that we would like to measure are subjective—that is, they are complex, qualitative, and involve human judgment or human performance.

- ***Software complexity is a psychological concept.*** If the term means anything, it deals with how complex the software is to a human. Indeed, the complexity metrics are sometimes explicitly referred to as measuring "psychological complexity."
- ***Goodness (efficiency, productivity, effectiveness) of testers is an issue of human performance.***
- ***"Extent of testing" metrics are judgment-driven.*** A theory of testing is embedded (often hidden) in such measures. We are not using the population of all possible tests of a product as our baseline when we compute code coverage, and therefore we can have a 100% covered product that still has undiscovered defects. What is the underlying theory and what does it tell us about the relationship between complete coverage and adequate testing?

It's not easy to measure subjective attributes. Simplistic notions of measurement simply will not work. In practice, they not only don't work—they can also create harmful side effects.

The Definition of Measurement

Within the computing literature, measurement is often defined as the "assignment of numbers to objects or events according to a clear cut rule."

This can't be the right definition. If it was, then we *could* measure "goodness of testers" by counting their bug reports. This (as we'll see below) is ridiculous. The problem is that there are lots of available clear-cut rules. We have to select *the right clear-cut rule.*

Therefore, we prefer the following definition:

> ***Measurement is the assignment of numbers to objects or events according to a rule derived from a model or theory.***[3]

Defining a Measure

To evaluate any proposed measure (or metric), we propose the following ten questions.

[3] There's no need to restrict measurement to the assignment of *numbers.* As the late S.S. Stevens pointed out, measurement can be looked at as a matching operation on two domains. For example, he would have people indicate the relative loudness of two tones by adjusting the relative brightness of two lights.

1. ***What is the purpose of this measure?*** Some measures are used on a confidential basis between friends. The goal is to spot trends, and perhaps to follow those up with additional investigation, coaching, or exploration of new techniques. A measure used only for this purpose can be useful and safe even if it is relatively weak and indirect. Higher standards must apply as the measurements become more public or as consequences (rewards or punishments) become attached to them.
2. ***What is the scope of this measure?*** Circumstances differ across groups, projects, managers, companies, countries. The wider the range of situations and people you want to cover with the method, the wider the range of issues that can invalidate or be impacted by the measure.
3. ***What attribute are we trying to measure?*** If you only have a fuzzy idea of what you are trying to measure, your measure will probably bear only a fuzzy relationship to whatever you had in mind.
4. ***What attribute: What is the natural scale of the attribute?*** We might measure a table's length in inches, but what units should we use for extent of testing?
5. ***What attribute: What is the natural variability of the attribute?*** If you measure two supposedly identical tables, their lengths are probably slightly different. Similarly, your weight varies a little bit from day to day. What are the inherent sources of variation of "extent of testing"?
6. ***What instrument are we using to measure the attribute and what reading do we take from the instrument?*** You might measure length with a ruler and come up with a reading (a measurement) of 6 inches.
7. ***What instrument: What is the natural scale of the instrument?*** Fenton & Pfleeger (1997) discuss this is detail.
8. ***What instrument: What is the natural variability of the readings?*** This is normally studied in terms of "measurement error."
9. ***What is the relationship of the attribute to the instrument?*** What mechanism causes an increase in the reading as a function of an increase in the attribute? If we increase the attribute by 20%, what will show up in the next measurement? Will we see a 20% increase? Any increase?
10. ***What are the natural and foreseeable side effects of using this instrument?*** If we change our circumstances or behavior in order to improve the measured result, what impact are we going to have on the attribute? Will a 20% increase in our measurement imply a 20% improvement in the underlying attribute? Can we change our behavior in a way that optimizes the measured result but without improving the underlying attribute at all? What else will we affect when we do what we do to raise the measured result? Austin (1996) explores the dangers of measurement—the unintended side effects that result—in detail, across industries. Hoffman (2000) describes several specific side effects that he has seen during his consultations to software companies.

The phrase "directness of measurement" is often bandied about in the literature. If "directness of measurement" means anything, it must imply something about the monotonicity (preferably linearity?) of the relationship between attribute and instrument as considered in both #9 and #10.

The phrases "hard measure" and "soft measure" are also bandied about a lot. Let me suggest that a measure is "hard" to the extent that we can describe and experimentally investigate a mechanism that underlies the relationship between an attribute and its measure and that accurately predicts the rate of change in one variable as a function of the other.

A simple example: Using a ruler

Let's start with an example of the simplest case, measuring the length of a table with a 1-foot ruler.

- ***Attribute.*** The attribute of interest is the length of the table.
- ***Attribute's Scale.*** We'll have more to say about scaling soon. For now, note that the length of the table can be measured on a ratio scale. A table that is 6 feet is twice as long as one that is 3 feet. If we multiplied the lengths (by 12 for example), the (6x12)=72 inch table would still be twice as

long as the (3x12)=36 inch table. This preservation of the ratio relationship ("twice as long") when both items are multiplied is at the essence of the ratio scale.

- ***Attribute's Variation.*** If you measure the lengths of a few tables, you'll get a few slightly different measurements because there is some (not much, but a little) manufacturing variability in the lengths of the tables. Tables that are supposed to be 6 feet long might actually vary between 5.98 and 6.02 feet. The length of individual tables might not vary much, but if you went to an office supply store and asked for a 6-foot table, the table that you would get would only approximate (perhaps very closely approximate) 6 feet.
- ***Instrument.*** The ruler is the measuring instrument. It's 1 foot long, so we'll have to lay it down 6 times to mark off a 6-foot length.
- ***Instrument's Scale.*** The instrument (ruler) measures length on a ratio scale.
- ***Instrument's Variation.*** Try to measure a 6-foot table with a 1-foot ruler ten times. Record your result as precisely as you can. You'll probably get ten slightly different measurements, probably none of them exactly 6.000 feet. Even with ruler-based measurement, we have error and variability.
- ***Theory of Relationship.*** The relationship between the attribute and the instrument is *direct*. They're both on the same scale and (except for random error) a change in the attribute results in a directly comparable change in the measured value. (Example: Cut the table down to 4 feet and the next time you measure it, the ruler measurement will be 4 feet.)
- ***Probable Side Effects.*** I don't anticipate any side effects of using a ruler to measure the length of the table. I don't think, for example, that that using rulers will encourage table manufacturers to change how they make tables.

Second example: The scaling problem

Suppose that Sandy, Joe and Susan run in a race. Sandy comes in first, Joe comes in second, and Susan third. The race comes with prize money. Sandy gets $10,000, Joe gets $1000 and Susan gets $100.

We know from the results that Sandy ran a faster race than Joe and that Joe ran a faster race than Susan. Let's review the theory of measurement associated with these results:

- ***Attribute.*** The attribute of interest is the speed of the runners.
- ***Attribute's Scale.*** The attribute's scale is probably best expressed in terms of miles (or meters) per hour. If so, this is a ratio scale (1 mile per 4 minutes = 15 miles per hour).
- ***Attribute's Variation.*** The race only gave us one sample of the speed of these runners. If they ran the same track and distance again tomorrow, they'd probably have slightly different times.
- ***Instrument.*** The instrument is the simple observation of the order of the runners as they cross the finish line. There are two different sets of markers on this instrument (like the inches and centimeter markers on rulers). One set of markers says 1st, 2nd, and 3rd. The other markers say $10,000, $1000, and $100. You might prefer to measure speed with some *other* instrument but (oops) you forgot your stopwatch and this is what you've got. (Please suspend judgment rather than quibbling with this detail of this example. I'm trying to use something that everyone understands in order to make the point easily. We run into plenty of dimensions that we have to measure with crude instruments because we have no idea what a better instrument, like a stopwatch, would be.)

- ***Instrument's Scale.*** This instrument operates on an ordinal scale, not a ratio scale and not an interval scale. The following comparisons should give you a sense of the differences among the scales.
  - If we were operating on a ***ratio scale***, we would be able to say that Susan (measured as 3) was three times slower than Sandy (measured as 1). Or we could say that Susan (measured as $100) was 1/100th as fast as Sandy (measured as $10,000). Probably, neither of these statements is correct and we have no basis for deciding which one is closer to the truth.
  - If we were operating on an ***interval scale***, we could say that the difference between Susan and Joe (3-2=1) was the same as the difference between Joe and Sandy (2-1=1) and that the difference between Susan and Sandy (3-1=2) was twice the difference between Susan and Joe.
  - On an ***ordinal*** (or positional) ***scale***, all we know is that Sandy crossed the line first, Joe crossed it second, and Susan crossed it third. Suppose that Bill crossed the finish line too, but was disqualified because he took steroids. It rarely makes sense to add or subtract numbers that are not at least interval-scaled. Here's another example of ordinal data--bug severity level. Is the difference between a Level 1 ("fatal") bug and a Level 2 ("serious") the same as the difference between a Level 2 and a Level 3 (moderate)?
  - The ordinal scale doesn't tell us much about speed but it tells us more than a ***nominal (or categorical) scale***. It would stretch the example too far to try to talk about a nominal scale for speed but since we've been comparing the different scale types here, this is the best place in this paper to describe a nominal scale. Suppose that you had hundreds of different pieces of cloth and you sorted them by color. Here is the brown pile. There is the pink pile. This is the green pile. By sorting, you have nominally (or categorically) scaled the cloths. For those who insist that measurement is the assignment of number to an attribute according to a clear cut rule, make brown color 1, pink color 2 and green color 3.
  - The final scale to mention is the ***absolute scale.*** If you have 1 child, you have 1 (one) child. One is 1 is one. Unlike inches and centimeters for length, the units of measurement (for number of children) are fixed. Two children are twice as many as one, and four are twice as many as two, but these ratios reflect more children, not a change in scale. You don't have two half-children, you have one child. If you cut the child in half, you have a mess, not 2 half-children and not (any longer) 1 child. (Too gruesome an example? Substitute "laptop computer" for child.)

Table 1: Examples of Scale Types
Absolute scale: You have four computers, I have two.
Ratio scale: You are twice as wealthy as me—you have $200, I have $100. The relationship is the same if we switch to Canadian dollars, $300CDN vs $150CDN. Multiplying doesn't affect the relationship.
Interval scale: Temperatures of 70, 75 and 80 degrees (Fahrenheit) differ from each other by 5 degrees. The difference (the interval) between 10 and 15, 70 and 75, and 75 and 80 is the same. But a 70 degree day is not 7 times as hot as a 10 degree day.
Ordinal: Bug severity: fatal, serious, moderate, minor.
Categorical: This is a design bug (category 1); that is a control flow bug (category 2).

- For extended definitions of the different scale types, see Stevens (1951, 1976) and Fenton & Pfleeger (1997) and Zuse (1997).

- ***Instrument's Variation.*** There is probably not much variation in this example, unless the race was very close. We would see variation if different judges, sitting in different places, disagreed as to whether Joe or Sandy came in first.
- ***Theory of Relationship.*** The mechanism underlying the relationship between speed and position in the race is straightforward. Faster speed results in a better position (first, second, third).
- ***Probable Side Effects.*** I don't see obvious side-effects (changing how people run races) in this particular case.

Example 3: Bug counts and the theory of relationship

Should we measure the quality (productivity, efficiency, skill) of testers by counting how many bugs they find? Leading books on software measurement suggest that we compute "average reported defects / working day" (Grady & Caswell, 1987, p. 227) and "tester efficiency" as "number of faults found per KLOC" (Fenton & Pfleeger, 1997, p. 36) or "defects found per hour" (Fenton, Pfleeger, & Glass, 1994). These authors are referring to averages, not measures of individual performance, and they sometimes warn against individual results (because they might be unfair.) However, I repeatedly run into pointy-haired bosses (or, at least, the test managers who work for them) who compute these numbers and take them into account for decisions about raises, promotions, and layoffs.

Let's do the analysis and see the problems with this measure:

- ***Attribute.*** The attribute of interest is the goodness (skill, quality, effectiveness, efficiency, productivity) of the tester.
- ***Attribute's Scale.*** I don't know. Neither do you. Suppose that one tester reports fewer bugs, but the ones reported are more subtle and required more investigation, or they are better analyzed and better described. Is this a better tester than one who reports many more bugs? *We can't measure "tester goodness" without a theory of tester goodness.*
- ***Attribute's Variation.*** I don't know. But there is variation. Joe probably does better work on some days than others.
- ***Instrument.*** We don't have an obvious, easy to use, unambiguous direct measure of tester effectiveness / skill / etc. So instead, we use a surrogate measure, something that is easy to count and that seems self-evidently related to the attribute of interest. In this case, the instrument is a counter of bug reports.
- ***Instrument's Scale.*** Bug counts are an absolute scale (two half-reports do not equal one whole-report).
- ***Instrument's Variation.*** There's not much random variation in counting of bug reports, though there is some, such as bugs classified as duplicates, changing the count. There is systematic measurement-related variation, as we'll see in the discussion of side effects.
- ***Theory of Relationship.*** There is a rough intuition that great testers report more bugs than crummy testers. But there is no theory of mechanism that says, an increase of so much in tester effectiveness will result in an increase of so many in the bug count. Here are two examples of disconnection between bug counts and value obtained from a tester. These are both from real situations.

  In the first case, corporate management was overeager to ship a product by a publicized release date. I assigned one tester to use his considerable exploratory testing skills to hunt for showstoppers. He reported as few as four bugs per month, compared to rates of perhaps 25-150 from the other eleven testers. We held that product for six months past the scheduled ship date by finding at least one showstopper per week, every week. Most weeks, one of those showstoppers (often the only one) came from our hunter. Was he the most productive member of our group or the least productive?

In the second case, a project team identified one area of the product (graphic filters) as very high risk. This was a new version of a hot selling mass market product and there were important business reasons for shipping the product on time. However, if we shipped with the serious problems in this area that we expected, we would be drowned in phone calls. The two testers assigned to this area found surprisingly few bugs and the project team (including me) worried that they might not have gotten a complete handle on the risks. I pulled in a very senior third tester who knew a lot about graphic filters to assess the testing and the area. Over about five weeks, he found about five bugs. He also reported that the test plans and strategies of the first two testers were pretty good. Partially based on this reassurance, we shipped the product a day ahead of schedule, sold a huge number of copies on the first day, and got few complaints.

In both of these cases, testers with very low bug counts did fine work and played a more significant role in the project than some other testers who found many more bugs in other areas. Bug counts would have seriously mis-measured these testers.

- ***Probable Side Effects.*** Here are just a few of the things that can and probably will go wrong if we measure individuals by counting their bug reports.
  - People are good at tailoring their behavior to things that they are measured against. If you ask a tester for more bugs, you'll probably get more bugs. Probably you'll get more bugs that are minor, or similar to already reported bugs, or design quibbles -- more chaff. But the bug count will go up. (Weinberg & Schulman, 1974)
  - People know that other people tailor their behavior. Put a tester under incentive to report more bugs and every programmer will become more skeptical of the value of the bug reports they receive. Does this tester believe in this bug, they ask, or is she just inflating her bug count? Bug counting creates political problems (especially if you also count bugs per programmer).
  - You can make a tester look good or bad just by choosing what type of testing she should do (regression testing often yields fewer bugs than exploratory testing of the same area of the program) or what area of the program she should test (fewer bugs to find in less buggy code). If raises and promotions are influenced by bug counts, project assignments will often be seen as unfair. More political problems.
  - The measurement system creates incentives for superficial testing (test cases are quick and easy to create) and against deep tests for serious underlying errors. Bug counts punish testers who take the time to look for the harder-to-find but more important bugs.
  - The system also penalizes testers who support other testers. It takes time to coach another tester, to audit his work, or to help him build a tool that will make him more effective. The tester who does this has less time to find bugs.
  - Time spent on any process that doesn't lead to more bugs faster is time that counts against the tester. For example, bug counting rewards testers who minimize the time they spend documenting their test cases.

  Problems like these have caused several measurement advocates to warn against measurement of attributes of individuals (e.g., Grady & Caswell, 1987) unless the measurement is being done for the benefit of the individual (for genuine coaching or for discovery of trends) and otherwise kept private (e.g. DeMarco, 1995). Often, people advocate using aggregate counts--but any time you count the output of a group and call that "productivity", you are making a personal measurement of the performance of the group's manager.

With only a weak theory of relationship between bug counts and tester goodness, and serious probable side effects, we should not use this measure (instrument).

Surrogate Measures

Surrogate measures provide unambiguous assignments of numbers according to rules, but they don't provide an underlying theory or model that relates the measure to the attribute allegedly being measured. As Mark Johnson (1996) put it (see also Austin, 1996):

> "Many of the attributes we wish to study do not have generally agreed methods of measurement. To overcome the lack of a measure for an attribute, some factor which can be measured is used instead. This alternate measure is presumed to be related to the actual attribute with which the study is concerned. These alternate measures are called surrogate measures."

Bug counts are classic examples of surrogate measures. We don't know how to evaluate the quality of a tester and so we count their bug reports. We get numbers, but they are meaningless, ripe for abuse, and more likely to do much more harm than good. But they play well with pointy-haired bosses, and if the point of measurement is to placate an executive who doesn't know better, they serve that purpose well.

Surrogate measures are at the root of an old corporate saying: ***There is nothing more dangerous than a vice-president with statistics.*** Be careful with what you feed these people. You might have to eat the results.

Exploring How To Measure Extent of Testing

When someone asks us for a report on the amount of testing that we've completed, what do they mean? The question is ambiguous.

At an early point in a class he teaches, James Bach has students (experienced testers) do some testing of a simple program. At the end of the exercise, James asks them (on a scale from 0 to 100, where 100 means completely tested) how much testing they've done, or how much they would get done if they tested for another 8 hours. Different students give strikingly different answers based on essentially the same data. They also justify their answers quite differently. We found the same broad variation at STMR (a collection of experienced test managers and test management consultants).

If someone asks you how much of the testing you (or your group) has done, and you say 30%, you might be basing your answer on any combination of the following dimensions (or on some others that I've missed):

- ***Product coverage:*** In principle, this compares the testing done with the total amount of testing of the product that could be done. But that can't be right because the amount of testing possible is infinite (Kaner, 1997) and 30% of infinity is unachievable. Instead, this percentage is offered with reference to a model (which might be implicit or explicit) of what would be good enough testing. Of the population of tests needed for sufficient testing, 30% have been done. ***This measure is inherently dynamic.*** At the start of testing, you might decide that certain tests need not be done. During testing, you'll probably discover that some of your assumptions were wrong. If someone finds bugs that you didn't expect or you learn something new about market expectations, you might decide to add new tests. If you do, then what you used to call 30% coverage might now be better estimated as 20%.
- ***Agreement-based:*** Compare to what has been promised (or planned and committed to). Given a list of tasks that you've agreed to do (for example, see Kaner, 1996), what percentage have you finished? However, if you add tasks or deepen your work in some areas, your percentage "complete" will need revision.
- ***Project-history based:*** Compare this project with some others that are similar or that can bracket your expectations for this one.
- ***Risk-based:*** One form of risk-based estimation translates into agreement-based estimation. You do a risk analysis of the product, derive a set of tests (or of areas to be tested, with time boxes for

the work), and then report what you've done against what you intended or agreed. (For example, see Amland, 1999.) A different estimator is less formal. When someone asks you how much you have gotten done, rather than saying 30%, you tell the person about the most critical areas of risk remaining in the product. Some areas are untested, others are undertested, others are awaiting difficult bug fixes. Assuming that nothing else bad comes up, you say, it will take about X weeks to deal with these, and then the product will probably be ready to release.

- ***Evaluation (of testing).*** This is another form of risk evaluation. How has the testing been? Are you finding surprises that indicate that your testing is weaker than expected? Should you reconsider your plans for the testing effort?
- ***Results:*** How much testing have you gotten done? For example, you might reports bug statistics or statistics on tests completed.
- ***Results and obstacles:*** When someone asks you how much testing you've actually gotten done, what is their purpose for collecting that information? Perhaps it is predictive (she wants an estimate of when the product can be shipped) or perhaps it is diagnostic or disciplinary. If the hidden question is not just "How far are you?" but is also "Why have you only gotten this far?" then your report of 30% might mean "30% of what was humanly possible to do during this period" and it might be accompanied by a list of obstacles that wasted your time or blocked some testing tasks.
- ***Effort:*** You report that you've tested for 400 hours or for 4 calendar weeks. Again, depending on the purpose of the question, this might be the best thing to report. Perhaps the underlying question is whether the you (or the testing group) are doing much work (or are able to focus your time on testing tasks). Or perhaps you work under a fixed budget or schedule, and you are reporting that you have consumed 30% of your resources. (If you are testing 10 projects at the same time, such measures can be particularly useful.)

Another Look at the Question

When we ask, "*What is a good measure of the extent of testing?*" what are we looking for? There are several related questions to consider:

- ***The measurement question:*** What are we trying to measure (what is the attribute) and how can we measure it?
- ***The communication question:*** How should we tell management how much testing is done and how much is left?
- ***The control question:*** Can we use measures of the extent of testing to help us control the testing project? Which measure(s) will be effective?
- ***The side effect question.*** What risks are inherent in measuring the extent of testing? In what ways might the process of measuring (and of managing on the basis of the measures) have results that we didn't intend or anticipate?
- ***The organizational effects question.*** In what ways might the use of this type of measure affect the organization?
- ***The requirements question:*** How does this type of measure serve our testing requirements? How do our requirements push us to choose measures?

These questions overlap with the measurement framework analysis but in discussions, they seem to stimulate different answers and different ideas. Therefore, for now, I'm keeping them as an independently useful list.

The Rest of this Paper

The material that follows lists and organizes some of the ideas and examples that we (see the Acknowledgement, above) collected or developed over the last two years. I have filtered out many suggestions but the lists that remain are still very broad. My intent is to show a range of thinking, to provide you with a collection of ideas from many sources, not (yet) to recommend that you use a particular measure or combination of them. The ordering and grouping of ideas here are for convenience. The material could be reorganized in several other ways, and probably in some better ways. Suggestions are welcome.

Let me stress that I do not endorse or recommend the listed measures. I think that some of them are likely to cause more harm than good. My objective is to list ideas that reasonable people in the field have found useful, even if reasonable people disagree over their value. If you are considering using one, or using a combination of them, I suggest that you evaluate the proposed measure using the measurement framework described above.

The sections below do not make a good enough connection with the software measurement literature. Several sources (for example, Zuse, 1997, and Ross Collard's software testing course notes) provide additional measures or details. As I suggested at the start, this paper reports work in progress. The next report will provide details that this one lacks.

Line/Branch Coverage Just Tests the Flow Chart

You're not testing

- data flow
- tables that determine control flow in table-driven code
- side effects of interrupts, or interaction with background tasks
- special values, such as boundary cases. These might or might not be tested.
- unexpected values (e.g. divide by zero)
- user interface errors
- timing-related bugs
- compliance with contracts, regulations, or other requirements
- configuration/compatibility failures
- volume, load, hardware faults

Coverage-Based Measures of Extent of Testing

"Coverage" is sometimes interpreted in terms of a specific measure, usually statement coverage or branch coverage. You achieve 100% statement coverage if you execute every statement (such as, every line) in the program. You achieve 100% branch coverage if you execute every statement and take every branch from one statement to another. (So, if there were four ways to reach a given statement, you would try all four.) I'll call this type of coverage (coverage based on statements, branches, perhaps also logical conditions), *code coverage.*

Code coverage is a tidy measure—it is easy to count, unambiguous, and easy to explain. Unfortunately, this measure carries risks (Marick, 1999). It is easy (and not uncommon) to write a set of relatively weak tests that hit all of the statements and conditions but don't necessarily hit them very hard. Additionally, code coverage is incomplete. For examples of incompleteness:

- Testing the lines of code that are there does not necessarily reveal the problems arising from the code that is not there. Marick (2000) summarizes data from cases in which 22% to 54% of the errors found were faults of omission.
- You can achieve 100% code coverage while missing errors that would have been found by a simple data flow analysis. (Richard Bender provides a clear and simple example of this in his excellent course on Requirements Based Testing.)
- Code coverage doesn't address interrupts (there is an implicit branch from every statement in the program to the interrupt handler and back, but because it is implicit—wired into the processor

rather than written into the program directly—it just doesn't show up in a test of every visible line of code) or other multi-tasking issues.

- Table-driven programming is puzzling for code coverage because much of the work done is in the table entries, which are neither lines of code nor branches.
- User interface errors, device incompatibilities, and other interactions with the environment are likely to be under-considered in a test suite based on code coverage.
- Errors that take time to make themselves visible, such as wild pointers or stack corruption or memory leaks, might not yield visible failures until the same sub-path has been executed many times. Repeatedly hitting the same lines of code and the same branches doesn't add any extent-of-testing credit to the code coverage measure.

Sometimes, the most important coverage measure has nothing to do with code coverage. For example, I worked on a product that had to print well. This was an essential benefit of the product. We selected 80 printers for detailed compatibility testing. We tracked the percentage of those 80 printers that the program could pass. This was a coverage measure, but it had nothing to do with lines of code. We might pass through exactly the same code (in our program) when testing two printers but fail with only one of them (because of problems in their driver or firmware).

You have a coverage measure if you can imagine any kind of testing that can be done, and a way to calculate what percent of that kind of testing you've done. Similarly, you have a coverage measure if you can calculate what percentage of testing you've done of some aspect of the program or its environment. As with the 80 printers, you might artificially restrict the population of tests (we decided that testing 80 printers was good enough under the circumstances) and compute the percentage of that population that you have run.

Here are some examples of coverage measures. Some of this list comes from Kaner (1995), which provides some additional context and discussion.

- ***Line coverage:*** Test every line of code (Or Statement coverage: test every statement).
- ***Coverage of non-dead code***, code that can be reached by a user.
- ***Branch coverage:*** Test every line, and every branch on multi-branch lines.
- ***N-length sub-path coverage:*** Test every sub-path through the program of length N. For example, in a 10,000 line program, test every possible 10-line sequence of execution.
- ***Path coverage:*** Test every path through the program, from entry to exit. The number of paths is impossibly large to test. (*See* Myers, 1979, and Chapter 2 of Kaner, Falk, and Nguyen, 1993).
- ***Multicondition or predicate coverage:*** Force every logical operand to take every possible value. Two different conditions within the same test may result in the same branch, and so branch coverage would only require the testing of one of them. (See Myers, 1979, for multiple condition coverage, and Beizer, 1990).
- ***Trigger every assertion check in the program:*** Use impossible data if necessary.
- ***Loop coverage:*** "Detect bugs that exhibit themselves only when a loop is executed more than once." (Marick, 1995, p. 146)
- ***Every module, object, component, tool, subsystem, etc.*** This seems obvious until you realize that many programs rely on off-the-shelf components. The programming staff doesn't have the source code to these components, so measuring line coverage is impossible. At a minimum (which is what is measured here), you need a list of all these components and test cases that exercise each one at least once.
- ***Fuzzy decision coverage.*** If the program makes heuristically-based or similarity-based decisions, and uses comparison rules or data sets that evolve over time, check every rule several times over the course of training.

- ***Relational coverage.*** "Checks whether the subsystem has been exercised in a way that tends to detect off-by-one errors" such as errors caused by using < instead of <=. (Marick, 1995, p. 147) This coverage includes:
  - *Every boundary on every input variable.* (Boundaries are classically described in numeric terms, but any change-point in a program can be a boundary. If the program works one way on one side of the change-point and differently on the other side, what does it matter whether the change-point is a number, a state variable, an amount of disk space or available memory, or a change in a document from one typeface to another, etc.? Kaner, Falk, & Nguyen, 1993, p. 399-401.)
  - *Every boundary on every output variable.*
  - *Every boundary on every variable used in intermediate calculations.*
- ***Data coverage.*** At least one test case for each data item / variable / field in the program.
- ***Constraints among variables:*** Let X and Y be two variables in the program. X and Y constrain each other if the value of one restricts the values the other can take. For example, if X is a transaction date and Y is the transaction's confirmation date, Y can't occur before X.
- ***Each appearance of a variable.*** Suppose that you can enter a value for X on three different data entry screens, the value of X is displayed on another two screens, and it is printed in five reports. Change X at each data entry screen and check the effect everywhere else X appears.
- ***Every type of data sent to every object.*** A key characteristic of object-oriented programming is that each object can handle any type of data (integer, real, string, etc.) that you pass to it. So, pass every conceivable type of data to every object.
- ***Handling of every potential data conflict.*** For example, in an appointment calendaring program, what happens if the user tries to schedule two appointments at the same date and time?
- ***Handling of every error state.*** Put the program into the error state, check for effects on the stack, available memory, handling of keyboard input. Failure to handle user errors well is an important problem, partially because about 90% of industrial accidents are blamed on human error or risk-taking. (Dhillon, 1986, p. 153) Under the legal doctrine of *forseeable misuse* (this doctrine is cleanly explained in Brown, 1991), the manufacturer is liable in negligence if it fails to protect the customer from the consequences of a reasonably forseeable misuse of the product.
- ***Every complexity / maintainability metric against every module, object, subsystem, etc.*** There are many such measures. Jones (1991, p. 238-341) lists 20 of them. Beizer (1990) provides a sympathetic introduction to these measures. Glass (1992) and Grady & Caswell, (1987) provide valuable perspective. People sometimes ask whether any of these statistics are grounded in a theory of measurement or have practical value. (*For example*, Kaner, Falk, & Nguyen, 1993, p. 47-48; *also* Glass, 1992, p. 303, "Software metrics to date have not produced any software quality results which are useful in practice.") However, it is clear that, in practice, some organizations find them an effective tool for highlighting code that needs further investigation and might need redesign. (*For example, see* Grady & Caswell, 1987, and Grady, 1992, p. 87-90.)
- ***Conformity of every module, subsystem, etc. against every corporate coding standard.*** Several companies believe that it is useful to measure characteristics of the code, such as total lines per module, ratio of lines of comments to lines of code, frequency of occurrence of certain types of statements, etc. A module that doesn't fall within the "normal" range might be summarily rejected (bad idea) or re-examined to see if there's a better way to design this part of the program.
- ***Table-driven code.*** The table is a list of addresses or pointers or names of modules. In a traditional CASE statement, the program branches to one of several places depending on the value of an expression. In the table-driven equivalent, the program would branch to the place specified in, say, location 23 of the table. The table is probably in a separate data file that can

vary from day to day or from installation to installation. By modifying the table, you can radically change the control flow of the program without recompiling or relinking the code. Some programs drive a great deal of their control flow this way, using several tables. Coverage measures? Some examples:

  - *check that every expression selects the correct table element*
  - *check that the program correctly jumps or calls through every table element*
  - *check that every address or pointer that is available to be loaded into these tables is valid (no jumps to impossible places in memory, or to a routine whose starting address has changed)*
  - *check the validity of every table that is loaded at any customer site.*

- ***Every interrupt.*** An interrupt is a special signal that causes the computer to stop the program in progress and branch to an interrupt handling routine. Later, the program restarts from where it was interrupted. Interrupts might be triggered by hardware events (I/O or signals from the clock that a specified interval has elapsed) or software (such as error traps). Generate every type of interrupt in every way possible to trigger that interrupt.
- ***Every interrupt at every task, module, object, or even every line.*** The interrupt handling routine might change state variables, load data, use or shut down a peripheral device, or affect memory in ways that could be visible to the rest of the program. The interrupt can happen at any time—between any two lines, or when any module is being executed. The program may fail if the interrupt is handled at a specific time. (Example: what if the program branches to handle an interrupt while it's in the middle of writing to the disk drive?) The number of test cases here is huge, but that doesn't mean you don't have to think about this type of testing. This is path testing through the eyes of the processor (which asks, "What instruction do I execute next?" and doesn't care whether the instruction comes from the mainline code or from an interrupt handler) rather than path testing through the eyes of the reader of the mainline code. Especially in programs that have global state variables, interrupts at unexpected times can lead to very odd results.
- ***Every anticipated or potential race.*** **(**Here as in many other areas, see Appendix 1 of Kaner, Falk, & Nguyen, 1993 for a list and discussion of several hundred types of bugs, including interrupt-related, race-condition-related, etc.) Imagine two events, A and B. Both will occur, but the program is designed under the assumption that A will always precede B. This sets up a race between A and B—if B ever precedes A, the program will probably fail. To achieve race coverage, you must identify every potential race condition and then find ways, using random data or systematic test case selection, to attempt to drive B to precede A in each case. Races can be subtle. Suppose that you can enter a value for a data item on two different data entry screens. User 1 begins to edit a record, through the first screen. In the process, the program locks the record in Table 1. User 2 opens the second screen, which calls up a record in a different table, Table 2. The program is written to automatically update the corresponding record in the Table 1 when User 2 finishes data entry. Now, suppose that User 2 finishes before User 1. Table 2 has been updated, but the attempt to synchronize Table 1 and Table 2 fails. What happens at the time of failure, or later if the corresponding records in Table 1 and 2 stay out of synch?
- ***Every time-slice setting.*** In some systems, you can control the grain of switching between tasks or processes. The size of the time quantum that you choose can make race bugs, time-outs, interrupt-related problems, and other time-related problems more or less likely. Of course, coverage is an difficult problem here because you aren't just varying time-slice settings through every possible value. You also have to decide which tests to run under each setting. Given a planned set of test cases per setting, the coverage measure looks at the number of settings you've covered.

- ***Varied levels of background activity.*** In a multiprocessing system, tie up the processor with competing, irrelevant background tasks. Look for effects on races and interrupt handling. Similar to time-slices, your coverage analysis must specify:
    - *categories of levels of background activity (figure out something that makes sense) and*
    - *all timing-sensitive testing opportunities (races, interrupts, etc.).*
- ***Each processor type and speed***. Which processor chips do you test under? What tests do you run under each processor? You are looking for:
    - *speed effects, like the ones you look for with background activity testing, and*
    - *consequences of processors' different memory management rules, and*
    - *floating point operations, and*
    - *any processor-version-dependent problems that you can learn about.*
- ***Every opportunity for file / record / field locking***.
- ***Every dependency on the locked (or unlocked) state of a file, record or field.***
- ***Every opportunity for contention for devices or resources.***
- ***Performance of every module / task / object:*** Test the performance of a module then retest it during the next cycle of testing. If the performance has changed significantly, you are either looking at the effect of a performance-significant redesign or at a symptom of a new bug.
- ***Free memory / available resources / available stack space at every line or on entry into and exit out of every module or object.***
- ***Execute every line (branch, etc.) under the debug version of the operating system:*** This shows illegal or problematic calls to the operating system.
- ***Vary the location of every file.*** What happens if you install or move one of the program's component, control, initialization or data files to a different directory or drive or to another computer on the network?
- ***Check the release disks for the presence of every file.*** It's amazing how often a file vanishes. If you ship the product on different media, check for all files on all media.
- ***Every embedded string in the program.*** Use a utility to locate embedded strings. Then find a way to make the program display each string.
- ***Every menu option.***
- ***Every dialogue or other data entry screen.***

Operation of every function / feature / data handling operation under:

- ***Every program preference setting.***
- ***Every character set, code page setting, or country code setting.***
- ***The presence of every memory resident utility (inits, TSRs).***
- ***Each operating system version.***
- ***Each distinct level of multi-user operation.***
- ***Each network type and version.***
- ***Each level of available RAM.***
- ***Each type / setting of virtual memory management.***

Compatibility

- ***Compatibility with every previous version of the program.***

- ***Ability to read every type of data available in every readable input file format.*** If a file format is subject to subtle variations (e.g. CGM) or has several sub-types (e.g. TIFF) or versions (e.g. dBASE), ***test each one.***
- ***Write every type of data to every available output file format.*** Again, beware of subtle variations in file formats—if you're writing a CGM file, full coverage would require you to test your program's output's readability by ***every one*** of the main programs that read CGM files.
- ***Every typeface supplied with the product.*** Check all characters in all sizes and styles. If your program adds typefaces to a collection of fonts that are available to several other programs, check compatibility with the other programs (nonstandard typefaces will crash some programs).
- ***Every type of typeface compatible with the program.*** For example, you might test the program with (many different) TrueType and Postscript typefaces, and fixed-sized bitmap fonts.
- ***Every piece of clip art in the product***. Test each with this program. Test each with other programs that should be able to read this type of art.
- ***Every sound / animation provided with the product.*** Play them all under different device (e.g. sound) drivers / devices. Check compatibility with other programs that should be able to play this clip-content.
- ***Every supplied (or constructible) script*** to drive other machines / software (e.g. macros) / BBS's and information services (communications scripts).
- ***All commands available in a supplied communications protocol.***
- ***Recognized characteristics.*** For example, every speaker's voice characteristics (for voice recognition software) or writer's handwriting characteristics (handwriting recognition software) or every typeface (OCR software).
- ***Every type of keyboard and keyboard driver.***
- ***Every type of pointing device and driver at every resolution level and ballistic setting.***
- ***Every output feature with every sound card and associated drivers.***
- ***Every output feature with every type of printer and associated drivers at every resolution level.***
- ***Every output feature with every type of video card and associated drivers at every resolution level.***
- ***Every output feature with every type of terminal and associated protocols.***
- ***Every output feature with every type of video monitor and monitor-specific drivers at every resolution level.***
- ***Every color shade displayed or printed to every color output device (video card / monitor / printer / etc.) and associated drivers at every resolution level.*** And check the conversion to grey scale or black and white.
- ***Every color shade readable or scannable from each type of color input device at every resolution level.***
- ***Every possible feature interaction between video card type and resolution, pointing device type and resolution, printer type and resolution, and memory level.*** This may seem excessively complex, but I've seen crash bugs that occur only under the pairing of specific printer and video drivers at a high resolution setting. Other crashes required pairing of a specific mouse and printer driver, pairing of mouse and video driver, and a combination of mouse driver plus video driver plus ballistic setting.
- ***Every type of CD-ROM drive, connected to every type of port (serial / parallel / SCSI) and associated drivers.***

- ***Every type of writable disk drive / port / associated driver.*** Don't forget the fun you can have with removable drives or disks.
- ***Compatibility with every type of disk compression software.*** Check error handling for every type of disk error, such as full disk.
- ***Every voltage level from analog input devices.***
- ***Every voltage level to analog output devices.***
- ***Every type of modem and associated drivers.***
- ***Every FAX command (send and receive operations) for every type of FAX card under every protocol and driver.***
- ***Every type of connection of the computer to the telephone line (direct, via PBX, etc.; digital vs. analog connection and signalling); test every phone control command under every telephone control driver.***
- ***Tolerance of every type of telephone line noise and regional variation (including variations that are out of spec) in telephone signaling (intensity, frequency, timing, other characteristics of ring / busy / etc. tones).***
- ***Every variation in telephone dialing plans.***
- ***Every possible keyboard combination.*** Sometimes you'll find trap doors that the programmer used as hotkeys to call up debugging tools; these hotkeys may crash a debuggerless program. Other times, you'll discover an Easter Egg (an undocumented, probably unauthorized, and possibly embarrassing feature). ***The broader coverage measure is every possible keyboard combination at every error message and every data entry point.*** You'll often find different bugs when checking different keys in response to different error messages.
- ***Recovery from every potential type of equipment failure.*** Full coverage includes each type of equipment, each driver, and each error state. For example, test the program's ability to recover from full disk errors on writable disks. Include floppies, hard drives, cartridge drives, optical drives, etc. Include the various connections to the drive, such as IDE, SCSI, MFM, parallel port, and serial connections, because these will probably involve different drivers.

Computation

- ***Function equivalence.*** For each mathematical function, check the output against a known good implementation of the function in a different program. Complete coverage involves equivalence testing of all testable functions across all possible input values.
- ***Zero handling.*** For each mathematical function, test when every input value, intermediate variable, or output variable is zero or near-zero. Look for severe rounding errors or divide-by-zero errors.
- ***Accuracy of every graph,*** across the full range of graphable values. Include values that force shifts in the scale.

Information Content

- ***Accuracy of every report.*** Look at the correctness of every value, the formatting of every page, and the correctness of the selection of records used in each report.
- ***Accuracy of every message.***
- ***Accuracy of every screen.***
- ***Accuracy of every word and illustration in the manual.***
- ***Accuracy of every fact or statement in every data file provided with the product.***

- ***Accuracy of every word and illustration in the on-line help.***
- ***Every jump, search term, or other means of navigation through the on-line help.***

Threats, Legal Risks

- ***Check for every type of virus / worm that could ship with the program.***
- ***Every possible kind of security violation of the program, or of the system while using the program.***
- ***Check for copyright permissions for every statement, picture, sound clip, or other creation provided with the program.***

Usability tests of:

- ***Every feature / function of the program.***
- ***Every part of the manual.***
- ***Every error message.***
- ***Every on-line help topic.***
- ***Every graph or report provided by the program.***

Localizability / localization tests:

- ***Every string.*** Check program's ability to display and use this string if it is modified by changing the length, using high or low ASCII characters, different capitalization rules, etc.
- ***Compatibility with text handling algorithms under other languages (sorting, spell checking, hyphenating, etc.)***
- ***Every date, number and measure in the program.***
- ***Hardware and drivers, operating system versions, and memory-resident programs that are popular in other countries.***
- ***Every input format, import format, output format, or export format that would be commonly used in programs that are popular in other countries.***
- ***Cross-cultural appraisal of the meaning and propriety of every string and graphic shipped with the program.***

Verifications

- ***Verification of the program against every program requirement and published specification.*** (How well are we checking these requirements? Are we just hitting line items or getting to the essence of them?)
- ***Verify against every business objectives associated with the program*** (these may or may not be listed in the requirements).
- ***Verification of the program against user scenarios.*** Use the program to do real tasks that are challenging and well-specified. For example, create key reports, pictures, page layouts, or other documents events to match ones that have been featured by competitive programs as interesting output or applications.
- ***Verification against every regulation (IRS, SEC, FDA, etc.) that applies to the data or procedures of the program.***

Coverage of specific types of tests:

- ***Automation coverage:*** Percent of code (lines, branches, paths, etc.) covered by the pool of automated tests for this product.
- ***Regression coverage:*** Percent of code that is covered by the standard suite of regression tests used with this program.
- ***Scenario (or soap opera) coverage:*** Percent of code that is covered by the set of scenario tests developed for this program.
- ***Coverage associated with the set of planned tests.***
  - Percent that the planned tests cover non-dead code (code that can be reached by a customer)
  - Extent to which tests of the *important* lines, branches, paths, conditions (etc.—important in the eyes of the tester) cover the total population of lines, branches, etc.
  - Extent to which the planned tests cover the *important* lines, branches, paths, conditions, etc.
  - Extent to which the scenario tests cover the *important* lines, branches, paths, conditions, or other variables of interest.

Inspection coverage

- ***How much code has been inspected.***
- ***How many of the requirements inspected/reviewed.***
- ***How many of the design documents inspected.***
- ***How many of the unit tests inspected.***
- ***How many of the black box tests inspected.***
- ***How many of the automated tests inspected.***
- ***How many test artifacts reviewed by developers.***

User-focused testing

- ***How many of the types of users have been covered or simulated.***
- ***How many of the possible use cases have been covered.***

Agreement-Based Measures of Extent of Testing

Agreement-based measures start from an agreement about what testing will and will not be done. The agreement might be recorded in a formal test plan or in a memo cosigned by different stakeholders. Or it might simply be a work list that the test group has settled on. The list might fully detailed, spelling out every test case. Or it might list more general areas of work and describe the depth of testing (and the time budget) for each one. The essence of agreement-based measures is progress against a plan.

All of these measures are proportions: amount of work done divided by the amount of work planned. We can convert these to effort reports easily enough just by reporting the amount of work done.

- ***Percent of test requirements turned into tests.***
- ***Percent of specification items turned into tests.***
- ***Percent of planned tests developed.***
- ***Percent of tests planned that have been executed.***
- ***Percent of tests executed that have passed.***

- ***Percent of tests passing, but that are old (haven't been rerun recently).***
- ***Percent (and list) of tests not yet executed.***
- ***Percent (list, whatever) of deliverables given to the client vs. total expected, where deliverables include bugs, scripts, reports and documentation (e.g. the matrix).***
- ***Test hours (or number of tests executed) per priority level (compared to the proportional levels of work intended for different priorities).***
- ***Test hours (or number of tests executed) per feature area (compared to the time intended for different areas).***
- ***Pattern of progress of testing (how much planned, how much done, how much left to do) across different test types across feature (or activity) areas.***
- ***Timesheet report of how tester hours were spent this week, against a pre-existing set of categories of tasks (time consumption types).***
- ***All test cases written / run.***
- ***Compare current results / status to product release criteria.***
- ***It's Tuesday – time to ship.***

Effort-Based Measures of Extent of Testing

These can be turned into agreement-based measures if we have an expected level of effort for comparison to the actual level completed.

- ***How much code must be tested.***
- ***How many spec pages / manual pages are associated with the material to be tested.***
- ***How many changes have been specified for the new version of an old product? How changes many are not specified but are being made?***
- ***How different is the new test plan from the old test plan.***
- ***Number of hours spent testing.***
- ***Amount of overtime per tester.***
- ***Compare the amount of overtime worked by testers and developers.***
- ***Number of test cycles completed.***
- ***Number of tests performed in each cycle.***
- ***Duration of each test cycle.***
- ***Number of tests executed on production hardware.***
- ***Number of test sessions (or some other unit of exploratory testing effort).*** See Bach (2000) for a discussion of exploratory testing sessions.
- ***Number of functional areas in which we're doing exploratory testing on a weekly basis.***
- ***Average cost of finding and/or fixing a bug.***
- ***Time spent maintaining automated tests {total, to date, per test cycle}.***
- ***Time to add new tests.*** How long it takes to add manual or automated tests to your population of tests. Include the time it takes to document the test, to file it in your source control system (if you have one), to plan the test, including checking whether it is redundant with current tests, etc.
- ***Time spent debugging the tests.***
- ***Time spent checking failures to see if you are reporting duplicates.***
- ***Time spent by testers writing defect reports.***

Project-History Based Measures of Extent of Testing

If you've run several projects, you can compare today's project with historical results. When someone claims that the project has reached "beta", you might compare current status with the state of the other products when they reached the beta milestone. For example, some groups add features until the last minute. At those companies, "all code complete" is not a criterion for alpha or beta milestones. But you might discover that previous projects were 90% code complete 10 weeks before their ship date.

Many of the measures flagged as based on risk, effort, or agreement can be used as project-history based if you have measured those variables previously, under circumstances that allow for meaningful comparison, and have the data available for comparison.

- ***How much time spent, compared to time expected from prior projects***
- ***Counts of defects discovered, fixed, open compared to previous project patterns.*** For example, number (or proportion) of defects open at alpha or beta milestone. These reports are widely used and widely abused. Significant side effects (impacts on the test planning, bug hunting, bug reporting, and bug triage processes) seem to be common, even though they are not often discussed in print. For a recent discussion that focuses on bug counts, see Hoffman (2000). For an excellent but more general discussion, see Austin (1996)..
- ***Arrival rates of bugs (how many bugs opened) per week, compared to rates on previous projects.*** I have the same concerns about the curves that show bugs per week as the single point (e.g. beta milestone) comparisons.
- ***How many (or percent of) modules or components coded compared to previous releases at this point in the project.***

Risk Based Measures of Extent of Testing

As I am using the term, the risk measures focus on risk remaining in the project. Imagine this question: *If we were to release the product today, without further testing, what problems would we anticipate? Similarly, if we were to release the product on its intended release date, what problems would we anticipate?*

- ***Of the test cases planned using hazard or risk analysis, how many have been written, run, or passed?***
- ***In a group that creates new tests throughout the testing project, have we reached a point in which we don't have any more questions?*** This might tell us that there is a problem with the test group, or it might tell us that we are close to being finished. (There might be plenty of errors left in the product, but if we're out of ideas, we're not going to find those errors.)
- ***How many bugs have been found in third party software that you include with or run with your products?***
- ***How many failures found per unit (per test case, per thousand pages printed, per thousand lines of code, per report, per transaction, per hour, etc.)***
- ***Mean time (transactions, etc.) between failures?***
- ***Length of time before system failure (or other detectable failures) in a stochastic test or a load or stress test.***
- ***Rate at which you are finding new bugs.***
- ***Rate at which previously fixed bugs are reappearing.***
- ***Number of complaints from beta testers.***
- ***Number of bugs found in the first manufactured copy of the product.*** If you release a product to manufacturing, you should test the first few manufactured copies, before sending the rest to

customers. Manufacturing errors happen. The disks (books, collaterals) that you sent out for duplication may or may not match what you get back.

- ***Number of bugs found in a random selection of manufactured copies.*** In most products, all of the manufactured copies are identical. Some products, however, vary from copy to copy. Even if the only difference across products is in the serial number, problems might show up on one copy that don't show up on others. The more extensively customized each copy is, the more relevant is this type of testing.
- ***After the product is released, we obtain data from the field that tell us whether we should reopen the product, such as:***
  - ***Angry letters to the CEO.***
  - ***Published criticism of the product.***
  - ***Rate of customer complaints for technical support.***
  - ***Number of "surprise" defects*** (problems found by customers that were not found pre-release by testers).
- ***Number of "surprisingly serious" defects*** (deferred problems that have generated more calls or angrier calls than anticipated.)
- ***Perceived risk.*** For example, the group might provide a rough estimate of risk for each area of the product on a 5-point scale (from low risk to very high risk), in order to think through the pattern of work left to be done in a short time available.
- ***Desperation level of triage team.*** The triage team is the group that decides whether or not to fix each defect. Toward the end of the project (especially if this "end" is a long time after the initially scheduled ship date), this team might be pushed very hard to stop fixing and just ship it. The more desperate the team is to ship the product *soon*, the higher the risk that the product will leave in bad shape.
- ***Number of defects resolved without being fixed.***
- ***Frequency of tester outbursts.*** This measure is informal but can be extremely important. As a poor quality product marches to release, many testers or test groups feel increasing pressure to find defects that will block the shipment. Frustration mounts as perfectly "good" bugs are deferred. Indication of high stress among the testers is a signal of danger in the product or the review process.
- ***Level of confusion or dissent among the development team as a whole or among sub-groups.*** (You measure this by talking with people, asking them how they feel about the product and the project, and its projected release date.

Obstacle Reports as a Measure of Extent of Testing

Obstacles are *risks to the testing project* (or to the development project as a whole). These are the things that make it hard to do the testing or fixing well. This is not intended as a list of everything that can go wrong on a project. Instead, it is a list of common problems that make the testing less efficient.

- ***Turnover of development staff (programmers, testers, writers, etc.)***
- ***Number of marketing VPs per release.*** (Less flippantly, what is the rate of turnover among executives who influence the design of the product?).
- ***Layoffs of testing (or other development) staff.***
- ***Number of organizational changes over the life of the project.***
- ***Number of people who influence product release, and level of consensus about the product among them.***

- ***Appropriateness of the tester to programmer ratio.*** (Note: I've seen successful ratios ranging from 1:5 through 5:1. It depends on the balance of work split between the groups and the extent to which the programmers are able to get most of the code right the first time.)
- ***Number of testers who speak English*** (if you do your work in English).
- ***Number of testers who speak the same language as the programmers.***
- ***How many bugs are found by isolated (such as remote or offsite) testers compared to testers who are co-located with or in good communication with the programmers?***
- ***Number of tests blocked by defects.***
- ***List of defects blocking tests.***
- ***Defect fix percentage (if low).***
- ***Slow defect fix rate compared to find rate.***
- ***Average age of open bugs.***
- ***Average time from initial report of defect until fix verification.***
- ***Number of times a bug is reopened (if high).***
- ***Number of promotions and demotions of defect priority***
- ***Number of failed attempts to get builds to pass smoke tests.***
- ***Number of changes to (specifications or) requirements during testing.***
- ***Percentage of tests changed by modified requirements.***
- ***Time lost to development issues*** (such as lack of specifications or features not yet coded).
- ***Time required to test a typical emergency fix.***
- ***Percentage of time spent on testing emergency fixes.***
- ***Percentage of time spent providing technical support for pre-release users (such as beta testers).***
- ***How many billable hours per week (for a consulting firm) or the equivalent task-focused hours (for in-house work) are required of the testers and how does this influence or interfere with their work?***
- ***Ability of test environment team (or information systems support team) to build the system to be tested as specified by the programming team.***
- ***Time lost to environment issues*** (such as difficulties obtaining test equipment or configuring test systems, defects in the operating system, device drivers, file system or other 3rd party, system software).

Evaluation-of-Testing Based Measures of Extent of Testing

These help you assess the testing effort. How hard are the testers doing, how well are they doing it, what could they improve? A high number for one of these measures might be good for one group and bad for another. For example, in a company that relies on an outside test lab to design a specialized set of tests for a very technical area, we'd expect a high bug find rate from third party test cases. In a company that thinks of its testing as more self-contained, a high rate from third parties is a warning flag.

- ***Number of crises involving test tools.***
- ***Number of defects related to test environment.***
- ***Number of bugs found by boundary or negative tests vs. feature tests.***
- ***Number of faults found in localized versions compared to base code.***

- ***Number of faults found in inspected vs. uninspected areas.***
- ***Number of defects found by inspection of the testing artifacts.***
- ***Number of defects discovered during test design (rather than in later testing).***
- ***Number of defects found by 3rd party test cases.***
- ***Number of defects found by 3rd party test group vs. your group.***
- ***Number of defects found by developers.***
- ***Number of defects found by developers after unit testing.***
- ***Backlog indicators.*** For example, how many unverified bug fixes are there (perhaps as a ratio of new bug fixes submitted) or what is the mean time to verify a fix?
- ***How many phone calls were generated during beta.***
- ***Number of surprise bugs (bugs you didn't know about) reported by beta testers.*** (the content of these calls indicate holes in testing or, possibly, weaknesses in the risk analysis that allowed a particular bug to be deferred)**.**
- ***After the product is released,***
  - *Number of "surprise" defects* (problems found by customers that were not found pre-release by testers).
  - *Number of "surprisingly serious" defects* (deferred problems that have generated more calls or angrier calls than anticipated.).
  - *Angry letters to the CEO*. (Did testers mis-estimate severity?)
  - *Published criticism of the product.* (Did testers mis-estimate visibility?)
  - *Rate of customer complaints for technical support*? (Did testers mis-estimate customer impact?)
- ***Number of direct vs. indirect bug finds (were you looking for that bug or did you stumble on it as a side effect of some other test?)***
- ***Number of irreproducible bugs (perhaps as a percentage of total bugs found).***
- ***Number of noise bugs (issues that did not reflect software errors).***
- ***Number of duplicate bugs being reported.***
- ***Rumors of off-the-record defects*** (defects discovered but not formally reported or tracked. The discoveries might be by programmers or by testers who are choosing not to enter bugs into the tracking system—a common problem in companies that pay special attention to bug counts.)
- ***Number of test cases that can be automated.***
- ***Cyclomatic complexity of automated test scripts.***
- ***Size (e.g. lines of code) of test automation code. Appraisal of the code's maintainability and modularity.***
- ***Existence of requirements analyses, requirements documents, specifications, test plans and other software engineering processes and artifacts generated for the software test automation effort.***
- ***Percentage of time spent by testers writing defect reports.***
- ***Percentage of time spent by testers searching for bugs, writing bug reports, or doing focused test planning on this project.*** (Sometimes, you're getting a lot less testing on a project than you think. Your staff may be so overcommitted that they have almost no time to focus and make real progress on anything. Other times, the deallocation of resources is intentional. For example, one company has an aggressive quality control group who publish bug curves and comparisons across

projects. To deal with the political hassles posed by this external group, the software development team sends the testers to the movies whenever the open bug counts get too high or the fix rates get too low.)

- ***Percentage of bugs found by planned vs. exploratory vs. unplanned methods, compared to the percentage that you intended.***
- ***What test techniques were applied compared to the population of techniques that you think are applicable compared to the techniques the testers actually know.***
- ***Comparative effectiveness:*** what problems were found by one method of testing or one source vs. another.
- ***Comparative effectiveness over time:*** compare the effectiveness of different test methods over the life cycle of the product. We might expect simple function tests to yield more bugs early in testing and complex scenario tests to be more effective later.
- ***Complexity of causes over time:*** is there a trend that bugs found later have more complex causes (for example, require a more complex set of conditions) than bugs found earlier? Are there particular patterns of causes that should be tested for earlier?
- ***Differential bug rates across predictors:*** defect rates tracked across different predictors. For example, we might predict that applications with high McCabe-complexity numbers would have more bugs. Or we might predict that applications that were heavily changed or that were rated by programmers as more fragile, etc., would have more bugs.
- ***Delta between the planned and actual test effort.***
- ***Ability of testers to articulate the test strategy.***
- ***Ability of the programmers and other developers to articulate the test strategy.***
- ***Approval of the test effort by an experienced, respected tester.***
- ***Estimate the prospective effectiveness of the planned tests.*** (How good do you think these are?)
- ***Estimate the effectiveness of the tests run to date*** (subjective evaluation by testers, other developers).
- ***Estimate confidence that the right quality product will ship on time*** (subjective evaluation by testers, other developers).

Results Reports as Measures of Extent of Testing

So what has the test group accomplished? Most of the bug count metrics belong here.

- ***Build acceptance test criteria and results.***
- ***Benchmark performance results.***
- ***Number of builds submitted for testing.***
- ***Number of builds tested.***
- ***Number of revisions (drafts) of the test plan***
- ***Number of test cases created / coded / documented.***
- ***Number of reusable test cases created.***
- ***Percent or number of reusable test artifacts in this project.***
- ***Number of times reusable artifacts are reused.***
- ***Number of defects in test cases.***
- ***List of failing (defective) tests.***
- ***Number of promotions and demotions of defect priority.***

- ***Tester to developer ratio.***
- ***Defects per 1000 lines of code.***
- ***Total bugs found / fixed / deferred / rejected.***
- ***Find and close rate by severity.***
- ***Total number of open bugs.***
- ***Number of open bugs by {feature, subsystem, build, database, supplier}.***
- ***Number of open bugs assigned to {programmer, tester, someone else, total}.***
- ***Number of open bugs by platform.***
- ***Defect find rate per week (open defects/week).***
- ***Cumulative defects each week (Total defects found/time.)***
- ***Defect find rate normalized by the amount of time spent testing.***
- ***Number of bugs found from regression testing.***
- ***Number of bugs fixed or failed from regression testing.***
- ***Number of bugs found in beta/real customer testing over time.***
- ***Percent of tests per build {passing, failing, not yet analyzed}.***
- ***Number of defects found in requirements / specifications / other development planning documents.***
- ***Number of defects found in unit testing.***
- ***Number of bugs found after a ship candidate has been declared***
- ***Number of defects found in LAN vs. modem testing. (Similarly for other configuration variations.)***
- ***Percentage of bugs found per project phase.***
- ***Number or percentage of bugs found by programmers.***
- ***Number of bugs found by programmers after unit testing.***
- ***Number of bugs found in first article (first item manufactured by manufacturing).***
- ***Proportion of unreviewed bugs that become critical after review. (Some companies don't have a review process. Others have each bug report reviewed by (depending on the company) another tester, a programmer, or the triage team as a whole.)***

Examples of Progress Reporting (Extent of Testing)

The following examples are based on ideas presented at LAWST and STMR, but I've made some changes to simplify the description, or because additional ideas came up at the meeting (or since) that seem to me to extend the report in useful ways. Occasionally, the difference between my version and the original occurs because (oops) I misunderstood the original presentation.

Feature Map

This description is based on a presentation by Jim Bampos and an ensuing discussion. This illustrates an approach that is primarily focused on agreement-based measures.

For each feature, list different types of testing that would be appropriate (and that you intend to do). Different features might involve different test types. Then, for each feature / testing type pair, determine when the feature will be ready for that type of testing, when you plan to do that testing, how much you plan to do, and what you've achieved.

A spreadsheet that tracked this might look like:

Feature	**Test Type**	**Ready**	**Primary Testing (week of. . .)**	**Time Budget**	**Time Spent**	**Notes**
Feature 1	*Basic functionality*	*12/1/00*	*12/1/00*	*½ day*		
	Domain	*12/1/00*	*12/1/00*	*½ day*		
	Load / stress	*1/5/01*	*1/15/01*	*1.5 days*		
	Scenario	*12/20/00*	*1/20/01*	*4 days*		

The chart shows "time spent" but not "how well tested." Supplementing the chart is a list of deliverables and these are reviewed for quality and coverage. Examples of primary deliverables are:

- Bug reports
- Test plans (test case design matrices, feature coverage matrices, etc.)
- Test scripts
- Summary reports (describing what was done / found for each area of testing)
- Recommendation memos.

Component Map

This description is based on a presentation by Elisabeth Hendrickson and an ensuing discussion. It primarily focuses on agreement-based measures.

For each component list the appropriate types of testing. (Examples: functionality, install/uninstall, load/stress, import/export, engine, API, performance, customization.) These vary for different components. Add columns for the tester assigned to do the testing, the estimated total time and elapsed time to date, total tests, number passed, failed or blocked and corresponding percentages, and the projected testing time for this build. As Elisabeth uses it, this chart covers testing for a single build, which usually lasts a couple weeks. Later builds get separate charts. In addition to these detail rows, a summary is made by tester and by component.

Component	**Test Type**	**Tester**	**Total Tests Planned / Created**	**Tests Passed / Failed / Blocked**	**Time Budget**	**Time Spent**	**Projected for Next Build**	**Notes**

Area / Feature Summary Status Chart

Gary Halstead and I used a chart like this to manage a fairly complex testing project. This was one of the successful projects that led to my report in Kaner (1996) rather than one of the failures that led to my suggesting in that report that this approach doesn't always work.

For our purposes, think of the output from testing project planning as a chart that spans many pages.

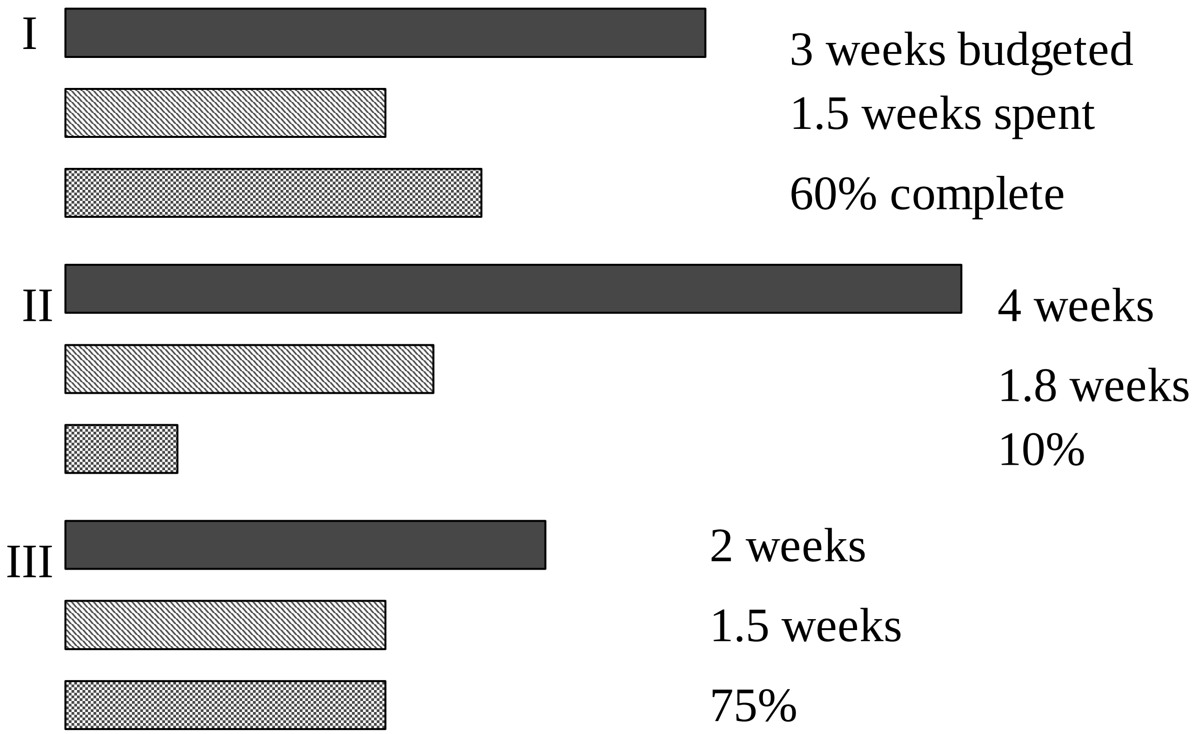
- An "area" represents a significant block of expenses. For example, "printer configuration testing" might be an area. An area might include one or more features or one type of testing or it might include a pure block of expense like “vacations and training.” If you will spend a lot of staff time on something, you can call it an area. I prefer to work with 10-25 areas.

- Within an area, there are sub-areas, such as individual features, or sub-features. Alternatively, you might break an area down into a series of different types of tests that you would apply within that area. Ideally, you’ll break the area down until you have a list of items that won’t take more than a day apiece. In some cases, you might not understand the task well enough to break it down this finely, or the natural breakpoint might involve more time. The goal is to end up with a list with tasks that are well enough broken down that you can make reasonable time estimates..
- For each sub-area, estimate the depth of testing to be done. We used labels like “mainstream”, “guerilla” (which means time-boxed exploratory testing), “formal planned testing,” and “planned regression testing” to identify different depths, but you can use any categories that work. “Small”, “medium” and “large” might be good enough.
- Next, estimate time. How long will it take to test this feature using this technique (do this sub-area task) at this level of depth of testing?
- The chart has two more columns, one to record how much time was actually spent and the other to record an estimate of how much work in the sub-area task was actually completed.

This chart might run as many as 50 pages (one row per sub-area task). No one will want to review it at a status meeting, but it is a useful data collection worksheet. There are various ways to figure out, each week, how much got done on each task. This administrative work is not inexpensive. However, it can pay for itself quickly by providing relatively early notice that some tasks are not being done or are out of control.

A *summary* of the chart is useful and well received in status reports. For each area, you can determine from the worksheets the total amount of time budgeted for the area (just add up the times from the individual tasks), how much time has actually been spent, and what percentage of work is actually getting done. (Use a weighted average—a 4 week task should have 4 times as much effect on the calculation of total percent complete as a 1 week task.)

The figure below illustrates the layout of the summary chart. The chart shows every area. The areas are labeled I, II, and III in the figure. You might prefer meaningful names. For each area, there are three bars. The first shows total time budgeted, the second shows time spent on this area, the third shows percentage of this area’s work that has been completed. Once you get used to the chart, you can tell at a glance whether the rate of getting work done is higher or lower than your rate of spending your budgeted time.



Project Status Memo

Some test groups submit a report on the status of each testing project every week. I think this is a useful practice.

The memo can include different types of information. Borrowing an analogy from newspapers, I put the bug counts (software equivalent of sports statistics) (the most read part of local newspapers) back several pages. People will fish through the report to find them.

The front page covers issues that might need or benefit from management attention, such as lists of deliverables due (and when they are due) from other groups, decisions needed, bugs that are blocking testing, and identification of other unexpected obstacles, risks, or problems. I might also brag about significant staff accomplishments.

The second page includes some kind of chart that shows progress against plan. Any of the charts described so far (the feature map, the component map, the summary status chart) would be useful on the second page.

The third page features bug counts. The fourth page typically lists the recently deferred bugs, which we'll talk about at the weekly status meeting.

Across the four pages (or four sections), you can fit information about effort, obstacles, risk, agreement-based status, bug counts, and anything else you want to bring to the project team's or management's attention.

Effort Report

When you assign someone to work on a project, you might not get much of their time for that project. They have other things to do. Further, even if they were spending 100% of their time on that project, they might only be running tests a few hours per week. The rest of their time might be spent on test planning, meetings, writing bug reports, and so on.

The following chart is a way of capturing how your staff spend their time.

Rather than using percentages, you might prefer to work in hours. This might help you discover that the only people who are getting to the testing tasks are doing them in their overtime. (Yes, they are spending 20% of their time on task, but that 20% is 10 hours of a 50 hour week.)

Effort-based reporting is useful when people are trying to add new projects or new tasks to your group's workload, or when they ask you to change your focus. You show the chart and ask which things to cut back on, in order to make room for the new stuff.

	0-10%	**10%-20%**	**20%-30%**	**30%-40%**	**40%-60%**
Coordination	8	2	0	2	0
Status Reporting	12	0	0	0	0
Setup Environment	4	3	3	0	1
Scoping	5	5	2	0	0
Ad Hoc	7	2	1	2	0
Design/Doc	1	3	5	1	2
Execution	1	3	5	3	0
Inspections	7	5	0	0	0
Maintenance	9	3	0	0	0
Review	10	2	0	0	0
Bug Advocacy	5	5	2	0	0

A Set of Charts for Requirements-Driven Testing

Jim Kandler manages testing in an FDA-regulated context. He tracks/reports the progress of the testing effort using a set of charts similar to the following four:

- Traceability matrix that traces test cases to requirements
- Test case status chart
- Testing effort charts
- Bug status charts

Traceability Matrix

In the traceability matrix, you list specification items or requirements items across the top (one per column). Each row is a test case. A cell in the matrix pairs a test case with a requirement. Check it off if that test case tests that requirement.

	Requirement 1	Requirement 2	Requirement 3	Requirement 4	Requirement 5	Etc.
Test 1	*		*			
Test 2		*	*		*	
Test 3			*			
Totals	1	1	3	0	1	

The chart shows several things:

- If Requirement 5 changes, then you know that you need to update test case 2 and that you probably don't need to update the others.
- Requirement 4 is untested
- Requirement 3 is tested in every test. The matrix is *unbalanced*—there appears to be too much testing of some features compared to others.

Notice that if we don't write down a requirement, we won't know from this chart what is the level of coverage against that requirement. This is less of an issue in an FDA-regulated shop because all requirements (claims) must be listed, and there must be tests that trace back to them.

Test Case Status Chart

For each test case, this chart tracks whether it is written, has been reviewed, has been used, and what bugs have been reported on the basis of it. If the test is blocked (you can't complete it because of a bug), the chart shows when the blocking bug was discovered.

Many companies operate with a relatively small number of complex regression tests (such as 500). A chart that shows the status of each test is manageable in size and can be quite informative.

Test Case ID	**Written**	**Reviewed**	**Executed**	**Blocked**	**Software Change Request**
1	1	1			
2	1	1	1		118
3	1	1	1	10/15/00	113
4	1				
5					
6					
7					
Totals 7	**4**	**3**	**2**		
100%	**57%**	**42%**	**29%**		

Testing Effort Chart

For each tester, this shows how many hours the person spent testing that week, how many bugs (issues) she found and therefore how many hours per issue. The numbers are rolled up across testers for a summary report to management that shows the hours and issues of the group as a whole.

Week of	Hours	Issues	Hr/Issue
20 Aug	19.5	19	1.03
27 Aug	40.0	35	
3 Sept	62	37	
10 Sept	15	10	

A chart like this helps the manager discover that individual staff members have conflicting commitments (not enough hours available for testing) or that they are taking remarkably short or long times per bug. Trends (over a long series of weeks) in availability and in time per failure can be useful for highlighting problems with individuals, the product, or the project staffing plan.

As a manager, I would not share the individuals' chart with anyone outside of my group. There are too many risks of side effects. Even the group totals invite side effects, but the risk might be more manageable.

Bug Status

Bug handling varies enough across companies that I'll skip the chart. For each bug, it is interesting to know when it was opened, how long it's been opened, who has to deal with it next, and its current resolution status (under investigation, fixed, returned for more info, deferred, etc.). Plenty of other information might be interesting. If you put too much in one chart, no one will understand it.

Summary

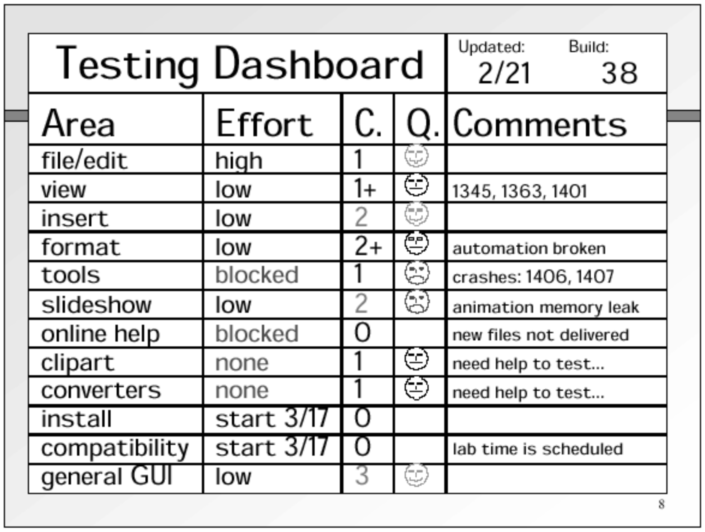
Jim shared some lessons learned with these four charts (traceability, test case status, effort, and bug statistics) that are generally applicable for project status reporting. In summary, he said:

- It doesn't have to be complicated to be valuable. Simple charts are valuable.
- It's important to understand your universe (or scope) of tests.
- It's important to have very clean tracking of progress against the universe of tests.
- Under the right circumstances, repeated use of the same tests can be very valuable.
- You learn a lot of project status issues by reporting multi-week statistics instead of a snapshot.
- It's valuable to track human hours against a specific activity. The process of tracking this leads you to questions that help you manage the project more effectively.
- You can pinpoint high risk areas once you've established scope.
- You can use this tool as a guide to the chronology of the project.
- This approach to tracking is time effective and easy to delegate.
- It's instructive to note that not all tests are created equal—the percentages in the chart may be misleading.

These lessons apply well in highly regulated industries. In mass-market software, some of them will be less applicable.

Project Dashboard

James Bach has spoken several times about his project status dashboard. See, for example, Bach (1999), which includes the following picture as an example. This is a simple tool, but a very useful one.



Testing Dashboard				Updated: 2/21 Build: 38
Area	Effort	C.	Q.	Comments
file/edit	high	1		
view	low	1+		1345, 1363, 1401
insert	low	2		
format	low	2+		automation broken
tools	blocked	1		crashes: 1406, 1407
slideshow	low	2		animation memory leak
online help	blocked	0		new files not delivered
clipart	none	1		need help to test...
converters	none	1		need help to test...
install	start 3/17	0		
compatibility	start 3/17	0		lab time is scheduled
general GUI	low	3		

8

The dashboard is created on a large whiteboard, typically in the main project conference room.

Different groups vary the columns and the headings. Bach prefers to keep the design as simple and uncluttered as possible. Here's the variation that I prefer (which breaks one of James' columns in two).

- ***Area:*** Break the testing project down into not more than 25 or 30 areas (at most).
- ***Effort***: How much work / time are we currently spending on this area? ("Blocked" means that we want to test in this area but we can't because the code isn't testable, perhaps because of missing code or a blocking bug.)
- ***Coverage planned***: However we define coverage, how much coverage do we intend to achieve in this area? (0 = untested, 3 = very high coverage).
- ***Coverage achieved:*** How much have we gotten done?
- ***Quality:*** Blank = no data. Frowny-face = bad quality. Neutral-face = not great yet but not a crisis. Smiley-face = high quality.
- **Comments:** Includes notes and bug report numbers. Whatever is needed to quickly flag issues.
- **Green means things go well. Red indicates a serious problem. Yellow indicates a problem that needs attention.**

A manager can read the testing project status at a glance from this board. If there are lots of frowny faces and red rows, the project has trouble. If planned and achieved coverage don't match, the product needs more testing. Update the dashboard frequently (every day or few days) and it becomes a focal point for project status discussions.

Bach (1999) provides more information about the dashboard in practice. His live presentation is worth attending.

Summing Up on Extent of Testing

Many test groups focus their progress reports on bug counts. When you ask how much testing they've done, they say, 322 bugs worth. Or they speak in terms of a product coverage measure, like code coverage.

There are many other ways to answer the questions:

- How much testing have you gotten done?
- How much do you have left to do?
- How are you going to tell management about it?

This paper has provided a survey of suggested answers, a framework for evaluating them, and samples of presentation approaches used by several successful test managers and consultants.

This paper doesn't provide what we might think that we really need—a small set of measures that are known to be useful and valid, and a standardized reporting style that everyone will anticipate and learn to understand.

It takes time to develop and validate a good set of metrics. I'm not there yet. I don't think the field is there yet. I don't expect us to be there next year or the year after, but we'll make progress. Unless we have a breakthrough, progress is incremental.

Measurement and Software Engineering

Over the past decade, there has been a strong political movement to license software engineers. For an example of this advocacy, see the November, 1999 issue of *IEEE Software*. Licensing carries consequences. If you are licensed in engineering, you can be sued for malpractice if your work fails to meet the level of competence expected of a member of the profession. In a trial, the decision as to whether given actions or decisions were malpractice is made by jurors (non-engineers), with the assistances of expert witnesses who testify as to the profession's standards.

We have many objections to the notion of licensing software engineers, and they start with the problems of measurement.

Engineering involves the application of scientific knowledge to solve practical problems. Much of that scientific knowledge is based on measurement.

Bill Hetzel (1996) tried to find out the basis of published software measurements. He asked direct questions of several leading people in the field.

- He was unable to obtain detailed descriptions of experiments underlying the results. This means that he cannot independently replicate the experiments.
- He was unable to obtain the reference data themselves. This means that he cannot independently reanalyze the results.
- He was not even able to get answers to questions like, "How was testing effort calculated and measured?" in response to claims like, "Inspections found 1 defect per hour of effort compared to testing that typically was .2 or .3 per hour."

Hetzel reported that much of the data that are available are from proprietary software or from toy programs. This certainly accords with our experience. And as a recent public example, Watts Humphrey presented some striking data at the QAI conference in Orlando, 1999. James Bach recently asked whether he could have access to the data and Watts said no, of course not, these data are proprietary. (We are pleased to be able to use Watts Humphrey as our example because we consider him a man of impeccable character. He illustrates a problem with the industry as a whole, not a problem that might be more personal.)

These problems are typical of applied research.

Research labs in the traditional sciences operated differently. It is not uncommon to reanalyze other researchers' data and to reinterpret their results. Data are expected to be open so that any motivated researcher can

(a) learn how to replicate the experiment,

(b) reanalyze the data to determine whether they have been fairly presented, and

(c) reanalyze the data to determine whether other interpretations are also supported.

Without this openness, the data are anecdotal, subject to no independent quality control. It is hard to see such data as providing a sound basis for claims that we have arrived at a "scientific" plateau that lays a fair basis for determining whether someone committed software engineering malpractice.

APPENDIX: HISTORY OF MEASUREMENT IN ANOTHER FIELD

This appendix is not included in the QUEST conference draft of this chapter. We have most of the text available but are still drawing the pictures. The Appendix traces a 150 year history of theory of measurement in Psychophysics. Psychological measurement is an interesting area for comparison with computing because so many of the attributes that we want to study (productivity, complexity, usability) heavily involve human performance.

References

Amland (1999), "Risk Based Testing and Metrics" *16th International Conference on Testing Computer Software.*

Austin, R.D. (1996) *Measuring and Managing Performance in Organizations*.

Bach, James (1999) "A Low Tech Testing Dashboard", *STAR Conference East*, Available at www.satisfice.com/presentations/dashboard.pdf.

Bach, Jonathan. (2000) "Measuring Ad Hoc Testing", *STAR Conference West*.

Beizer, B. (1990) *Software Testing Techniques* (2nd Ed.)

Boring, E.G. (1917) "A chart of the psychometric function" *American Journal of Psychology*,*28*, 465-470.

Boring, E.G. (1943) *Sensation and Perception in the History of Experimental Psychology.* Appleton Century Crofts.

S. Brown (Ed., 1991) *The Product Liability Handbook: Prevention, Risk, Consequence, and Forensics of Product Failure*.

Cornett, S., *Code Coverage Analysis*, www.bullseye.com/coverage.html

DeMarco, T. (1995), "Mad about measurement" in *Why Does Software Cost So Much?*, Dorset House.

Dhillon, B.S. (1986) *Human Reliability With Human Factors*.

Fenton, Norman E., Shari L. Pfleeger, & R.L. Glass (1994) "Science and substance: A challenge to software engineers," *IEEE Software*, 11(4), 86-95.

Fenton, Norman E. & Shari L. Pfleeger (1997, 2nd Ed.) *Software Metrics: A Rigorous & Practical Approach*, PWS Publishing.

Fenton, Norman E., Whitty & Iizuka (1996) *Software Quality Assurance & Measurement: A Worldwide Perspective*, Thomson Computer Press.

Gescheider, G. (1997) *Psychophysics: The Fundamentals*, 3rd Ed., Lawrence Erlbaum.

Glass, R.L. (1992) *Building Quality Software*.

Grady, R.B. (1992) *Practical Software Metrics for Project Management and Process Improvement*.

Grady, R.B. & D.L. Caswell (1987) *Software Metrics: Establishing a Company-Wide Program*, PTR Prentice-Hall.

Green, D.M. & J.A. Swets (1966) *Signal Detection Theory and Psychophysics*, Wiley.

Hetzel, Bill "The sorry state of software practice measurement and evaluation" in Fenton, Whitty & Iizuka (1996).

Hoffman, D. (2000) “The darker side of metrics”, *Pacific Northwest Software Quality Conference*.

Johnson, M.A. (1996) *Effective and Appropriate Use of Controlled Experimentation in Software Development Research*, Master's Thesis (Computer Science), Portland State University.

Jones, C. (1991) *Applied Software Measurement*, 1991, p. 238-341.

Kaner. C. (1983) Auditory and Visual Synchronization Performance Over Long and Short Intervals, Doctoral Dissertation, McMaster University, Department of Psychology.

Kaner, C. (1995) “Software negligence and testing coverage”, *Software QA Quarterly*, Volume 2, #2, 18. Available at www.kaner.com.

Kaner, C. (1996) “Negotiating testing resources: A collaborative approach”, *Ninth International Software Quality Week Conference.* Available at www.kaner.com.

Kaner, C. (1997) “The impossibility of complete testing”, *Software QA*, Volume 4, #4, 28. Available at www.kaner.com.

Kaner, C., J. Falk, & H.Q. Nguyen (1993, reprinted 1999) *Testing Computer Software* (2nd. Ed.), Wiley.

Kitchenham, Pfleeger, & Fenton (1995) “Towards a framework for software measurement and validation.” *IEEE Transactions on Software Engineering*, vol. 21, December, 929.

Marick, B. (1995) *The Craft of Software Testing*.

Marick, B. (1999), “How to Misuse Code Coverage”, *16th International Conference on Testing Computer Software*. Available at www.testing.com/writings.html.

Marick, B. (2000), “Faults of Omission”, *Software Testing & Quality Engineering*, January issue. Available at www.testing.com/writings.html.

Myers, G. (1979) *The Art of Software Testing*.

Popper, Karl (1953) “Conjectures and refutations” in Karl Popper (1992) *Conjectures & Refutations: The Growth of Scientific Knowledge*, Routledge.

Schneidewind, N. (1994) “Methodology for Validating Software Metrics,” *Encyclopedia of Software Engineering* (Marciniak, ed.) *see also* IEEE 1061, *Standard for a Software Quality Metrics Methodology*.

Stevens, S.S. (1951) "Mathematics, measurement and psychophysics," in S.S. Stevens, (Ed., 1951) *Handbook of Experimental Psychology*, Wiley, 1-49.

Stevens, S.S. (1975) *Psychophysics*, Wiley.

Thurstone, L.L. (1959) *The Measurement of Values*, University of Chicago Press.

Urban, F.M. (1930) “The future of psychophysics”, *Psychological Review,37*, 93-106.

Weinberg, G.M. (1993) *Quality Software Management, Volume 2, First-Order Measurement*.

Weinberg, G.M. & E.L. Schulman (1974) "Goals and performance in computer programming," *Human Factors*, 16(1), 70-77.

Zuse, H. (1997) *A Framework of Software Measurement*.