

A Measured Response

Useful metrics to
give managers the
numbers to back up
project hunches

BY MIKE COHN

ARE YOU A DEVELOPMENT, PROJECT, OR TEST MANAGER?
THINK YOU HAVE YOUR PROJECT UNDER CONTROL? LET ME
ASK YOU A FEW SIMPLE QUESTIONS.

- When will the software be ready for use? Really? How do you know?
- How many more test cycles should we plan for? No...I mean *exactly*.
- Are the customer's needs being met? Prove it.
- We need to focus our efforts on code most likely to have errors. Which code would that be?

Feel like you are on the hot seat? Can you answer these questions with complete confidence? When the pressure is on to provide hard data to support your estimates, you need metrics. An infinite number of metrics can be applied to various aspects of software development. In fifteen years of managing software development, I've found a handful of metrics that really help me do my job—and keep me cool and confident when the heat is on.

Product Stabilization Metrics

Early in my career as a project manager, the CEO stopped me in the hallway and asked how long it would take for our new product to be ready for release. I told him “three or four weeks,” which seemed reasonable since we'd been testing for two weeks already and things were going well so far. A few years earlier, I'd learned the lesson that a date range estimate is always better than an exact date estimate. So I was pretty happy with my “three or four weeks,” rather than “June 13.” Then the CEO asked me how I knew it wouldn't be two weeks or five weeks. I didn't have a really good answer and didn't want to admit to the CEO that I was running his large project on “gut feel.”

Tracking defect inflow and outflow is one of the best ways to back up your gut when forecasting a product release. Defect inflow is the rate at which new bugs are reported against the system. On most products, defect inflow will look like an inverted V (see Figure 1, on the next page) because testing is slow going at first. Some of the reasons for this could be

1. The program contains “blocking bugs.” A blocking bug is a defect that prevents testing from occurring past the point of the bug. I once worked with a programmer who was offered tickets to a New York Knicks game if QA found no more than five bugs in the first version he turned

QUICK LOOK

- Tracking defect inflow and outflow
- Calculating defect comebacks
- Monitoring customer satisfaction
- Code review metrics

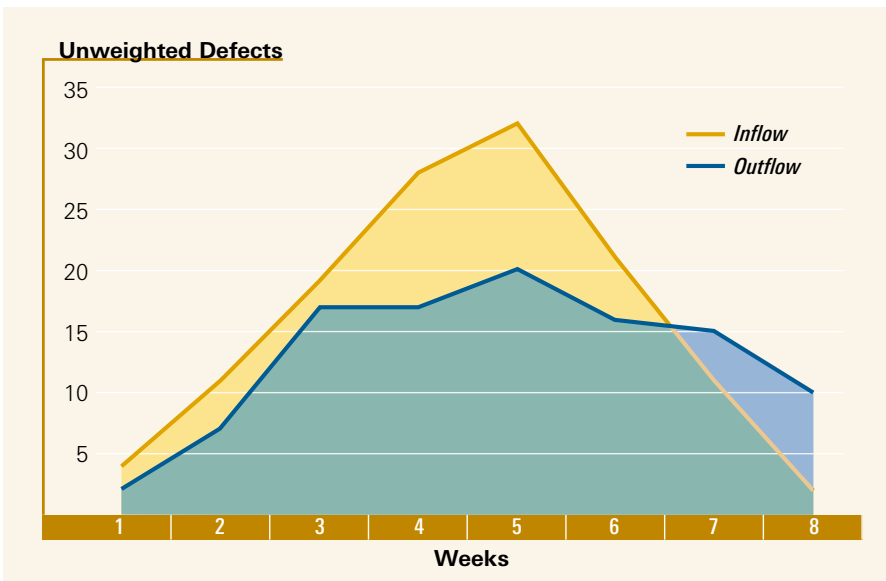


Figure 1: Number of bugs found and fixed each week

over to them. The version he first gave QA included a blocking bug with the log-on screen that prevented any access to the program whatsoever. That one bug blocked all further testing. Fortunately, our boss was smart enough not to give him the Knicks tickets, despite his program having only one bug.

2. Because testing just finished their last project, they are still trying to develop test plans for this one.
3. Testers are unfamiliar with the application because of changes to product designs or because testers were not involved with development discussions. There will be a learning curve.

Defect outflow represents the number of bugs fixed each week. Typically this will start out slow (it cannot exceed the rate of inflow at first, until a backlog of bugs builds), but will grow and level off at a steady rate. Just prior to release, there is usually a push to “fix only absolutely critical bugs.” At this point, defect outflow drops dramatically. The outflow line in Figure 1 is representative of most projects.

It is best to graph defect inflow and outflow on a weekly basis; daily graphing has too much variability. For example, if your best tester takes a day off, the graph will likely give the illusion that the product is improving. In terms of reports

to upper management, I’ve found it best to limit charts to the last eight periods of data. If you are using an agile process such as XP or Scrum, a project can change drastically in that time, so older data is of minimal value. Eight weeks is also long enough to see most trends. As you monitor defect inflow and outflow, consider all factors that may influence your measurements. You don’t want to make a pronouncement that the program is nearly shippable just because you see inflow coming down. Inflow may be coming down because testers currently are reworking a significant number of tests rather than doing hands-on testing.

By weighting defect inflow and outflow, you can get a MUCH BETTER FEEL for what is happening on a project... trends in weighted inflow usually lead trends in unweighted inflow.

Since all bugs are not equal, you may want to try weighted inflow and outflow tracking. Use a five-tier defect categorization and weighting such as

Critical	20
High	10
Medium	7
Low	3
Very Low	1

By weighting defect inflow and outflow, you can get a much better feel for what

is happening on a project. For example, I have typically found that trends in weighted inflow usually lead trends in unweighted inflow. On most projects, there is a point at which testing is still finding a significant number of bugs, but the bugs are lower in severity. In this case, you’ll see weighted inflow drop while unweighted inflow will typically remain constant. In most cases, this indicates that you are one to three periods away from seeing a matching drop in unweighted inflow. By measuring weighted defect inflow, you’ll typically be able to forecast this important turning point in a project a few weeks earlier.

Monitoring weighted outflow (the weighted defects fixed) helps ensure that your programmers are fixing the bugs you most want fixed. Regardless of how you define Critical and Low, you would almost assuredly prefer that programmers work on Critical defects. By discussing weighted outflow with programmers, you can direct their attention toward higher-value work.

Programmer Quality Metrics

Back in my pre-programming days when I was a mechanic, my boss introduced me to a car repair metric that translated perfectly into the software world: the comeback. To a mechanic, a comeback is a car that you supposedly fixed, but that later “comes back” with the same problem.

Some defects, like some cars, come back. It is useful to track defect come-

backs—or what I call “defect recidivism” when I want to use bigger words. You can calculate recidivism on the programming team as a whole, on each programmer individually, or on both.

Be careful when using recidivism metrics. In almost all cases, you should not use them as a significant factor in formal or informal evaluations of programmers. Consider an extreme case where you announce that a \$15,000 bonus will be paid to the programmer with the lowest recidivism rate on a

project. You may find yourself paying this bonus to a programmer who wrote very little code, tested it more than was appropriate, and who turned it over to QA late. That isn't typically behavior you'll want to encourage.

Knowing a team's recidivism rate is extremely useful in predicting when a product WILL BE READY FOR RELEASE.

Additionally, if recidivism is a significant factor in evaluating programmers, you will discourage them from working fixes in unfamiliar code. No one will volunteer to address a five-year-old bug, because it's likely to come back.

Individual recidivism rates can also vary tremendously because of the different tasks assigned to programmers. On a recent project, one of my best programmers had one of the highest recidivism rates because he was working on tens of thousands of lines of low-quality code written by his former coworkers; but he was fixing bugs at twice the rate of the team. It would have been unfair to compare his recidivism rate to others' on the same project.

Be careful in using metrics about individual programmers. Use the recidivism metrics as a clue that you need to investigate. For example: Does the developer need some additional skills training? Has she run into especially buggy code? Is she under severe schedule pressure and working sixteen hours a day and becoming too fatigued to contribute appropriately to the project?

With all these caveats in mind, should you measure individual recidivism rates? Absolutely. I'm not a big fan of writing quarterly or annual goals for individual programmers, but most HR departments insist on them. I give programmers annual goals about improving their recidivism rates. Once you've established a baseline recidivism rate (say 8%, meaning 8% of the bugs he "fixes" come back), you establish a goal for the programmer to strive for over the next period (perhaps 5%). Most programmers respond very positively, as they understand the impact that comebacks can have on project success.

Some programmers have no idea what their recidivism rates are. It is not uncommon to have a team with many programmers in the 5–8% recidivism

range, with one or two programmers at 15–20% or higher. These higher-recidivism programmers are unaware that they are that far away from their peers. For these teams, I may post weekly metrics showing recidivism rates for each pro-

grammer. I never comment on the metrics, but the programmers look at them and, in most cases, those with the highest comeback rates take the initiative to be more careful.

Team recidivism rates can support arguments against corporate mandates dic-

tating sixty-hour work weeks and other practices that lead to poorly crafted programs. If you can demonstrate with real metrics captured on your own projects that pushing people too hard results in more bugs (that then take longer to fix), you stand a reasonable chance of eliminating such policies.

Knowing a team's recidivism rate is extremely useful in predicting when a product will be ready for release. Assume that a product currently has 1,000 defects reported against it and, for simplicity, the test group is not looking for any new bugs. A team with a 5% recidivism rate should be able to release this product after the third pass through the test suite. A team with a 20% recidivism rate will

Counting Defect Inflow and Outflow

Counting defect inflow and outflow is complex. One common complication is how to count defects that have been reopened. For example, suppose QA reports a defect one day; a programmer marks it as fixed the next. Now, two weeks later, the bug shows up again. Should it be reported as part of inflow? I say no. Think of defect inflow as "defects first reported during the period." This is the easiest way to generate the metrics out of the defect tracking systems I've used.

In most defect tracking systems, it is easy to get a count of defects that were marked fixed in a given reporting period, but it is difficult to get a count of defects closed this week that had been closed before. Imagine the query "count all bugs that were opened this week or bugs reopened this week after being closed in a previous week."

Keep in mind, though, that if you count defect inflow as "defects first reported during the period" and outflow as "any bug marked as closed during the period," you can find yourself with some unusual results. For example, in a two-week period assume that one defect is reported during the first week and no defects are reported during the second week. If a programmer "fixes" the bug the first week, you'll show an outflow of one. During the second week, a tester discovers that the bug was only hidden and is really still there. If the programmer really fixes the bug the second week, you'll show no inflow but an outflow of one in that second week. In other words, your aggregate metrics will show one bug reported, two fixed.

Don't go through the extra work of reducing inflow or outflow counts from previous weeks. The metrics should reflect knowledge at the time they were collected. There will always be some amount of uncertainty in your latest period of metrics, so it is consistent to allow that same amount of uncertainty in prior weeks.

The real solution to situations like this is to make sure that anyone who will make decisions based on your metrics understands how they are calculated and any biases they may contain. For example, the method of counting inflow and outflow described above is what I call a "programmer-biased metric." In other words, the metric is completely consistent with how a programmer (as opposed to a tester) would view the counts. As such, it is likely to result in a slightly optimistic view of the program at any point. It is optimistic because defect inflow is shown as slightly less than it probably is and defect outflow is shown as slightly better than it probably is. Be sure that those who use your metrics understand the biases built into them.

	5% RECIDIVISM	20% RECIDIVISM
Bugs found during initial test runs	1,000	1,000
Comeback bugs from initial test runs	50	200
Comeback bugs from previous round	2.5	40
Comeback bugs from previous round	0	8
Comeback bugs from previous round	0	1.6
Comeback bugs from previous round	0	0

Figure 2: Effect of different recidivism rates on the test effort

likely take five passes (see Figure 2).

This type of calculation is useful when we are trying to schedule handoffs from programmers to QA and we need to know how many times a project will need to be tested. On a recent project, we had 106 bugs remaining and a 12% recidivism rate. In each cycle, QA found 1 or 2 new bugs. We determined that we would send the project back to QA after 80 fixes were in place. Of those 80, we expected 12% (9.6 defects) to come back, giving us 9.6 plus 2 newly discovered bugs plus 26 bugs that we had not yet attempted to fix. With 38 defects left and a 12% recidivism rate, we decided one more pass through QA would not be enough. We expected that we'd be left with about 5 comebacks plus 1 or 2 new bugs. We were confident, though, that with only 6 or 7 bugs left after two cycles, that these could be fixed in one final pass. By using recidivism this way, we were able to plan our effort to include three deliveries to QA and avoid a frantic last weekend.

Customer Satisfaction Metrics

If your customers aren't happy, then you won't be happy for long. One excellent use of metrics is to monitor customer satisfaction. Or, even better, to anticipate changes in customer satisfaction.

Robert Grady has introduced a very useful customer satisfaction metric that he calls "hotsites" and "alertsites." An alertsite or hotsite is a customer who is having a significant business issue with your software, not customers who are experiencing minor problems for which there are workarounds. You can define "significant" however you'd like in the context of your business: in a five-tier de-

fect classification, a good starting point would be the two highest severities.

In Grady's view, a customer who reports a significant problem is initially counted as an alertsite. That customer is moved to hotsite status "if no workaround is found quickly." I find this distinction too arbitrary. I use alertsite to refer to a customer who has reported a significant problem that has not yet been verified. A hotsite is a customer with a significant problem that *has* been verified.

Since not all hotsites are the same, weight each hotsite by the number of

Most development teams should do some code reviewing.
However, code reviews require a HUGE INVESTMENT OF TIME.

weeks (or possibly days) that the site has been hot. Create a hotsite index by summing this value for each hotsite. For example, a site reported a problem four weeks ago; you were able to verify the problem two weeks ago; but it still isn't fixed. This site would contribute four to the hotsite index. (Even though the site was only an alertsite for two weeks, it is important to capture that the customer has been impacted for four weeks.)

With multiple customer issues, indexes can climb quite dramatically. If you have a hotsite index of fifty today, comprised of sixteen different customers, you will have a hotsite index of sixty-six next week, unless some sites are eliminated. Even the elimination of two hotsites will probably not reduce the overall hotsite index next week. So once you start measuring customer satisfaction this way, it will take a significant amount of time to lower your alertsite and hotsite indexes.

I introduced these metrics into a software company with ten distinct prod-

ucts. They began the project thinking their customers were satisfied. Weekly alertsite and hotsite indexes for each product and for the overall department demonstrated there was significant room for improvement. This led to a weekly meeting where lead developers met with the CEO to discuss how to lower the indexes. It took about six months to eliminate all alertsites and hotsites. We then set a goal to never exceed a department aggregate hotsite or alertsite index of five. Once the indexes were down, we actually found it easy to keep them that way. Any time a customer issue came in, we immediately addressed it—probably with the urgency we should have had before introducing the metrics.

Complexity Metrics

In 1992, I was a programmer on a project that was committed to holding code inspections on absolutely every line of code. We met every day at 1:00 and spent at least an hour reviewing one programmer's code. The product was released successfully, did exactly what it was supposed to do, had no bugs reported from users, and went on to win a "product of the year"

award in its industry. I'm sure a big part of this success was the result of the time and effort we put into the code inspections.

I don't think the effort was worth it.

We spent too much time inspecting code that had no errors. We probably didn't spend enough time inspecting code that had the most errors. Instead of inspecting every line of code, we should have inspected only the code most likely to have errors. One problem: we didn't know how to identify that code.

Since that project, I've learned the value of code complexity metrics in identifying which code most needs to be inspected. The assumption is that if code is overly complex, it will probably have more than its share of defects. Even if the code isn't defect-prone today, it may need to be modified to reduce the complexity so defects don't appear as soon as a maintenance programmer gets involved with that code.

Most development teams should do some code reviewing. However, code re-

```

1. void foo(int a, int b, int c) {
2.     b += 100;
3.     if(a == b) {
4.         c = a + b;
5.         puts("a and b are equal");
6.     }
7.     else if (a > b) {
8.         while(c < a) {
9.             puts(".");
10.            c++;
11.        }
12.    }
13.    else
14.        c = -1;
15.    printf("c = %d\n", c);
16. }

```

Figure 3: Simple C-language function

views require a huge investment of time. This investment can be dramatically reduced with metrics, such as cyclomatic complexity or Halstead Difficulty. These metrics calculate aspects of complexity that can be used to identify code that could benefit most from inspection. Less complex code is reserved for a less time-intensive process, such as peer review.

Cyclomatic Complexity Metric

Thomas McCabe introduced the cyclomatic complexity metric in 1976. McCabe constructed a program control graph of the program and applied concepts of graph theory to it to measure “cyclomatic complexity.”

While there are many ways to calculate cyclomatic complexity, the easiest way is to sum the number of binary decision statements (e.g., if, while, for, etc.) and add 1. Count the statements in Figure 3:

1. **if** statement on line 3
2. **else-if** on line 7
3. **while** on line 8

(Note that the else statement on line 13 is not counted because it is not a separate decision; it is part of the decision made on lines 3 and 7.)

Since there are three decisions, the cyclomatic complexity is 3+1=4. I like this method because it is fairly easy to write a program to do this work for you. (For more information on cyclomatic complexity calculation methods, see the StickyNotes at the end of this article.)

Halstead Method

Maurice Halstead proposed his own measure of complexity, via a thorough examination of a program’s operators and operands. Operators are words in a program that do things—method calls, mathematical functions, etc. Operands are the items given to an operator. So, a/b has two operands (a and b) and one operator (the division symbol). By looking at a program’s operators and operands, and the relationships between them, Halstead measured attributes such as the length, volume, and difficulty of a program. Each of these higher-level metrics is calculable from the following four parameters:

- n_1 —the number of unique operators
- n_2 —the number of unique operands
- N_1 —the number of total operators
- N_2 —the number of total operands

The metric was intended to capture the difficulty of reading, writing, or maintaining the program, represented by D. D is given by the following equation:

$$D = (n_1 / 2) \cdot (N_2 / n_2)$$

Other methods exist to calculate complexity. Many commercially available source code analysis programs and tools generate values for cyclomatic complexity, Halstead Difficulty, and many other source code metrics for most common programming languages. (See the StickyNotes for more information.)

However you calculate it, code becomes too complex when cyclomatic complexity ratings are in the ten to fifteen range, or the Halstead Difficulty reaches sixty. Start with a cyclomatic complexity threshold of ten and see how much code that gives you to inspect. If it seems you’re still spending too much time inspecting, set the threshold at fifteen and try that. If you see too many error-prone modules making it to QA, set the threshold lower. The threshold you select will depend on many factors, including how much time you have to invest in code reviews and how experienced your engineers are with both the language and the domain.

A Parting Concern

With any measurement effort, some people will adjust behavior to measure well against the metric. For example, if you institute a policy of reviewing all methods with a cyclomatic complexity greater than

fifteen, most of your programmers will try to avoid crossing that threshold. Some will do it the way you want—by writing less complex code. Others will just break their complex code into multiple methods, thereby spreading the complexity (but not getting rid of it) and avoiding detection.

There are two good methods for managing this human tendency to alter behavior once it becomes measured. First, you can make it very clear that there is no consequence to the measurement. This was my recommendation with programmer defect recidivism. I don’t take the worst couple of programmers out behind the corporate woodshed; I just want to identify them so I can provide additional training, put them on more appropriate tasks, or just be aware of the issue when scheduling.

The second method is to put in place a metric that will counteract the behavior. For example, I’ve mentioned the value of giving programmers goals to reduce personal recidivism rates. Whenever I give that goal to a programmer, I make sure I pair it with a concurrent goal related to completing certain programming tasks within appropriate time frames.

At least a minimal metrics effort should be a part of every significant software development effort. Without metrics, you are forced to rely on gut feel or other intangible or anecdotal evidence in answering many questions about your projects. The needs of each project or development organization are different, and you should use the metrics suggested in this article as a starting point in thinking about and instituting your own metrics effort. *STQE*

Mike Cohn (mike@mountaingoatsoftware.com) has twenty years of experience developing software and is the president of Mountain Goat Software (www.mountaingoatsoftware.com). He authored four books on Java and C++ development. Mike is also actively involved with the Agile Alliance (www.agilealliance.com).

STQE magazine is produced by STQE Publishing, a division of Software Quality Engineering.