The Darker Side of Metrics

Douglas Hoffman, BACS, MBA, MSEE, ASQ-CSQE
Software Quality Methods, LLC.
24646 Heather Heights Place
Saratoga, California 95070-9710
doug.hoffman@acm.org

Abstract

There sometimes is a decidedly dark side to software metrics that many of us have observed, but few have openly discussed. It is clear to me that we often get what we ask for with software metrics and we sometimes get side effects from the metrics that overshadow any value we might derive from the metrics information. Whether or not our models are correct, and regardless of how well or poorly we collect and compute software metrics, people's behaviors change in predictable ways to provide the answers management asks for when metrics are applied. I believe most people in this field are hard working and well intentioned, and even though some of the behaviors caused by metrics may seem strange, odd, or even silly, they are serious responses created in organizations because of the use of metrics. Some of these actions seriously hamper productivity and can effectively reduce quality.

This paper focuses on a metric that I've seen used in many organizations (readiness for release) and some of the disruptive results in those organizations. I've focused on three different metrics that have been used and a few examples of the behaviors elicited in organizations using the metrics. For obvious reasons, the examples have been altered to protect the innocent (or guilty).

Biography

Douglas Hoffman is an independent consultant with Software Quality Methods, LLC. He has been in the software engineering and quality assurance fields for over 25 years and now is a management consultant specializing in strategic and tactical planning for software quality. He is Section Chairman for the Santa Clara Valley Section of the American Society for Quality (ASQ) and is past Chairman of the Silicon Valley Software Quality Association (SSQA). He is also a member of the ACM and IEEE, and is certificated by ASQ in Software Quality Engineering and has been a registered ISO 9000 Lead Auditor. He has earned an MBA as well as an MS in Electrical Engineering and BA in Computer Science. He has been a speaker at dozens of software quality conferences including PNSQC and has been Program Chairman for several international conferences on software quality.

The Darker Side of Metrics[1,2]

Introduction

Software measures and metrics have been around and used since the earliest days of programming. I have studied and used software measures and metrics with varying degrees of success throughout my career. I might even be labeled a reformed measurement enthusiast[3]. During the 25 years or so that I have studied and used software metrics I have been surprised by some of the effects the metrics have had on the organizations, and often I have been extremely distressed over the negative impacts I have seen. Even though I have touted software metrics and successfully begun several metrics programs, every software organization I have observed that has used metrics for more than a few years has had bizarre behaviors as a result. There is a decidedly "dark side" to these metrics programs that impacts organizations all out of proportion to what is intended. In the last year Kaner[4,5,6] has provided a framework for understanding why this might occur. One source comes from a lack of relationship between the metrics and what we want to measure (Kaner's 9th factor)[7] and a second problem is the over-powering side effects from the measurement programs (Kaner's 10th factor)[8]. The relationship problem stems from the fact that the measures we are taking are based on models and assumptions about system and organizational behavior that are naïve at best, and more often just wrong[9]. Gerald Weinberg provides excellent examples in his *Last Word* article analyzing some benign software inspection metrics[10]. Weinberg shows how counting defects found during preparation and at code inspections gives metrics relating mostly to the number of inspectors and telling almost nothing about the product or process it proports to measure.

It is clear to me that we often get what we ask for with software metrics. Whether or not our models are correct, and regardless of how well or poorly we collect and compute software metrics, people's behaviors change in predictable ways to provide the answers management asks for when

---

[1] This information was first generated for presentation and discussion at the *Eighth Los Altos Workshop on Software Testing* in December, 1999. I thank the LAWST attendees, *Chris Agruss, James Bach, Jaya Carl, Rocky Grober, Payson Hall, Elisabeth Hendrickson, Bob Johnson, Mark Johnson, Cem Kaner, Brian Lawrence, Brian Marick, Hung Quoc Nguyen, Bret Pettichord, Melora Svoboda,* and *Scott Vernon,* for their participation and ideas.

[2] I differentiate between the measures of an attribute and metrics computed from the measures. Ultimately we should take measures to compute metrics.

[3] Lawrence, Brian "Measuring Up," *Software Testing and Quality Engineering* vol. 2, no. 2 (2000)

[4] Kaner, C. "Rethinking Software Metrics," *Software Testing and Quality Engineering* vol. 2, no. 2 (2000)

[5] Kaner, C. "Yes, But What Are We Measuring?," 1999 PNSQC

[6] Kaner, C. "Measurement of the Extent of Testing," 2000 PNSQC

[7] ibid.

[8] ibid.

[9] Many models go so far as to ignore mathematical truths. Many times we categorize based on ordinal scales; Defect Severity, for example. We assign numbers to the categories and depict the order based on the values we chose. We know that a "Severity 1" isn't ½ as much as a "Severity 2," and we can't claim that all "Severity 3" defects are the same. We could just as well use colors and call the categories as Green, Yellow, Orange, and Red. Doing arithmetic with them (e.g., the Priority is Severity times Likelihood) is as absurd as multiplying colors.

[10] Weinberg, G. "How Good Is Your Process Measurement," *Software Testing and Quality Engineering* vol. 2, no. 1 (2000)

metrics are applied. Don't take me wrong; I believe most people in this field are hard working and well intentioned. Although some of the behaviors cause by metrics may seem funny or even silly, there are potentially serious consequences to organizations because they use metrics. The specific observations I make here are based on real companies using software metrics in their product development. I have taken some care to change enough of the details that the innocent (or guilty) cannot be easily identified. In some instances, I have combined observations from multiple organizations. But, you wouldn't be alone if you think you recognize your organization in some of the situations. I've noticed that people often recognize their own experiences here.

Three Metric Examples

I've selected three examples of metrics used to decide when a product is ready to release. There certainly are other examples and other metrics, but this has been a particularly ripe area of examples from my experience. The three metrics used to show a product's readiness for release are:

1. Defect find/fix rate
2. Percent of tests running/Percent of tests passing
3. Complex model based metrics (e.g., COCOMO)

Briefly, each of the metrics is used to describe an attribute of project status (how far along is the project, is it ready for release, are we meeting our milestones, etc.). These attributes were applied by management to monitor and adjust project plans and member behaviors in order to keep the project on schedule. I haven't a clue about the attributes' scales and don't think anyone else can, either. The variation in the attribute and the measures is all over the map – a few projects run like clockwork (or so I've heard), but most don't run as planned, and some I've worked with were just out of control. (Out of control is a term I use for

Kaner's Ten Measurement Factors

1. The purpose of the measure. What the measurement will be used for.
2. The *scope of the measurement*. How broadly the measurement will be used.
3. The *attribute to be measured*. E.g., a product's readiness for release.
4. The appropriate *scale for the attribute.* Whether the attribute's mathematical properties are rational, interval, ordinal, nominal, or absolute.
5. The *natural variation of the attribute.* A model or equation describing the natural variation of the attribute. E.g., a model dealing with why a tester may find more defects on one day than on another.
6. The *instrument that measures the attribute.* E.g., a count of new defect reports.
7. The *scale of the instrument*. Whether the mathematical properties of measures taken with the instruments are rational, interval, ordinal, nominal, or absolute.
8. The *variation of measurements* made with this instrument. A model or equation describing the natural variation or amount of error in the instrument's measurements.
9. The *relationship between the attribute and the instrument*. A model or equation relating the attribute to the instrument.
10. The *probable side effects* of using this instrument to measure this attribute. E.g., changes in tester behaviors because they know the measurement is being made.

software that has progressively much worse quality as developers try to patch it up, followed by project cancellation or quick turnover of most of the management and staff.)

I've neaver heard of any direct measure of project status, program readiness for release, or progress toward meeting milestones, etc. Instead, we've used surrogate measures and metrics; the instruments we used to measure are:

1) counters of new and resolved defect reports,
2) percents of tests running and passing, and
3) a "Mulligan's stew" of metrics (including cyclomatic complexity, defect counts, defect find/fix rates, defect severities, estimated size of programs, experience levels of developers, past projects' metrics, and others) combined and mixed thoroughly in an arithmetic equation (such as COCOMO).

Defect find/fix rate

The first two metrics use pairs of measures to determine convergence on the planned project completion. The ratio of defects found to defects fixed intuitively feels like a reasonable way to see the end. When we find more than we fix (ratio greater than 1) during a specified time period, we are discovering problems faster than fixing them. When the ratio equals 1, we are not gaining ground or losing it in terms of fixing problems. When the ratio gets below 1, the developers are reducing the number of known problems. For the life of the project, the ratio of all defects found to all defects fixed should approach 1 as we fix all known problems. This model is based on several assumptions that don't hold:

1) all defects that are found are reported,
2) there is a goal of fixing (or resolving) all known defects,
3) when all known defects are fixed the product is ready to release,
4) there are reasonable resolutions for all fixed defects.

Defect counts don't naturally vary, given a consistent definition of defects. One has either been found and reported or it hasn't. The count of the number of reported defects can easily be done, and several people are likely to come up with the same counts given the same defect database. However, human nature makes hash of the numbers by accident and with purpose. Reducing the effort to hunt new defects or withholding reports will directly and immediately improve the ratios. Developers and testers can become extremely creative in recategorizing defects as enhancement requests, problems in other products, duplicates, unassigned, etc. in order to resolve them without fixing the underlying problems.

A few examples of observed behaviors of these sorts may serve to clarify:

- To reduce the number of defects, twenty-five reports against a subsystem were all marked as "duplicates" of one new defect. The new defect report referred to each of the twenty-five for a

description of the problem (because the only thing the twenty-five had in common was that they were reported against the same subsystem).

- In an organization where defects didn't get counted before initial screening and assignment, a dozen defects that hadn't been resolved in more than four weeks were assigned to the developer "Unassigned," and thus were not counted.
- In one case the testers withheld defect reports to befriend developers who were under pressure to get the open defect count down. In another case the testers would record defects when the developers were ready with a fix to reduce the apparent time required to fix problems.
- A test group took heat for not having found the problems sooner (to give the developers more time to fix the problems).
- Developers only reported problems after they had been fixed (thus never making the ratio worse).
- I've seen defects fall in the crack, get lost, pushed in circles, or be forever deferred.

One general reaction to found/fixed metrics was the creation of "Pocket lists" of defects by developers and testers. Developers kept these unofficial lists of defects and action items to themselves. If they reported the defects, it created a negative perception about the code and they also needed to address the problems. One manager went so far as to publicly criticize individuals for having more than five defects against their modules. The project (with 40 modules) now *never* has more than 200 defects reported (and seldom fewer than that). The pocket list has been as benign as not reporting problems observed and fixed in code, and as blatant as knowing about existing problems others were likely to encounter.

There are also ways that management can make this situation much worse (such as a Dilbertian "bug bonus" for the number of bugs found or fixed), so the developer and tester are encouraged to report and fix large numbers of defects kept in pocket lists so they create the appearance of a flurry of last moment heroic activity. Indeed, it is quite difficult to take software measures without creating significant side effects.

Percent of tests running/Percent of tests passing

The second case (percent of tests running and percent passing) also may intuitively feel like a reasonable way to see the end. If 100% of our tests run, and 100% pass, we're done, right? The percentage of tests running can be interpreted two ways; either the testers haven't had time to run all the tests, or the software isn't complete enough to fully test. Likewise, the percent of tests that pass may feel like a reasonable way to measure progress. The terms themselves are difficult to pin down, and no matter how they are defined and enforced, there are simple ways to manipulate the percentages. Also, this model of testing makes several false assumptions:

1) all tests are known before testing begins,
2) whatever constitutes "a test" is well understood and agreed upon,
3) whatever constitutes "running a test" is well understood and agreed upon,
4) test outcomes are clearly pass or fail, and

5) release occurs when all tests (or a specified percentage) pass.

Very few organizations I've worked with do only pre-defined tests. Most test groups mix regression tests with exploration and continue to create new tests until the product escapes. Unless we know the exact number of tests we will run, we don't know the denominator. And, the definitions of a test, running a test, and passing and failing are subject to debate and manipulation. If a test is used in several configurations, is it a separate "test" in each? If it's only counted as a test once, do failures in several (but not all) configurations count as one failure, several, or a fraction of one? Does a test fail twice if it detects more than one defect? Does a long exercise count as multiple tests? Do we only count those tests that are for completed features? Do we have to run tests we know will fail? Do we have to report defects against those failures?

Some examples of observed behaviors due to these metrics:

1) the redefinition of what a "test" is in order to increase the number to be counted and increase the percentage passing. (Each test is divided into sections because having one of ten (1000 line) tests that can't run looks much worse than three of two hundred (50 line) tests.)
2) top management declares victory and releases because "All four of the tests that could run were tried, and 100% passed. (The code just wasn't complete for the other 5,768 tests.)"
3) replacement of expected results with actual (bad) results because a problem was "known." (Development demanded that testing remove the test from the count of tests running and not passing for those defects that weren't going to be fixed. Management would not let the testers reduce the count of tests running, so they compromised…) [I call this "institutializing a defect" – making sure it stays in the product forever.]

Complex model based metrics

The last situation uses complex multivariate mathematical models to describe the project. A "mathematical" model is applied to decide ahead of time how many defects there should be in modules. The models are also used to predict the rate of defect reporting, so the progress and readiness of a project can be discerned simply by counting the defects found to date. "According to the model, we should find 50% of the defects through our testing. Since there are 200 defects in the project (according to the Function Point computations), the project should be ready to release when we've found 100 defects." These predictions then become self-fulfilling prophecies, with sometimes crippling side effects.

For these measures, there isn't any real relationship between the measured or subjectively assigned attributes and the meaning assigned. There might be a relationship between the mathematical model and organizational behavior or project status, but I doubt it. The amazing thing I've observed is the near religious fervor that goes into defending the validity of the model based on the fact that on the last *N* projects, the equation has yielded a precise estimate of release on the day of release. (Never mind that it wasn't correct any of the 52 weeks previous to that, and the subjective values and equation fudge factors were changed every one of the past 52 times the equation was used.) This is what I call the "you always find your keys at the last place you look for them" effect. When they look back at the project,

they conclude they now know the numbers for "experience factors," "product complexity," and all the other elements that plug into the equation. They only really know that numbers can be chosen for the equation to show what they now know – its ready for release. The sad thing is the number of managers and engineers who don't realize that given any moderately complex equation with multiple variables, values can be selected to generate any particular result. For example, given the equation 5 * X + 3 * Y + Z, we can pick values of X, Y, and Z to yield any positive result.

Some examples of observed behaviors due to these metrics:

- Managers demanding sign off by testers without testing because the model showed release should occur in spite of testing not being complete. ("We've followed the curve precisely for eight months – obviously it's ready for release.")
- Punishment of testers for not finding enough defects quickly enough through lowering their rating (and thus their pay).
- Reporting of minor problems, variations on a defect, and seriously questionable tests, to increase defect counts.
- Testers not reporting defects (or reporting trivial problems) in order to keep the defect counts corresponding with the predicted values from the model. (Because management had such faith in the models, people were expected to perform exactly as predicted. Having too many defects was interpreted as meaning the project was poor quality and too few was interpreted as the tester doing a poor job.) [I've been given the argument that "the exactly predicted number of defects was reported every week on an 18 month project." This is a statistical impossibility for a real world process. W. Edwards Demming described the real world effect as being due to normal random variation. My chemistry instructors called too perfect data in an experiment "dry-labbing," documenting the predicted results instead of taking accurate measurements (even subconsciously).

Other Side Effects

In response to noticing some of the above side effect behaviors, the rules can change. In some situations, defect fixes cannot be deferred to future releases because opening a project with defects already reported skews the statistics (and anyway, the argument goes, the defects weren't put in on this project; they were put in on previous projects). But, since products cannot be released with known defects and the defects cannot be deferred, they are resolved ('not a defect' or 'no fix') in the current project and may be reentered if someone remembers them during the next project.

Deferral was used as a technique to reduce the number of defects, so management mandated justification in person for all deferred defects. Consequently, the number was reduced through consolidation of defects (25 marked as duplicates of a new one that references each of the 25). Then all duplicates required management review. Defects were then resolved en mass as "no fix intended." The rules changed then to force management review of all defect reclassification. Then defects became stacked into "submitted/need assignment" to keep them from getting into the statistics until resolution was ready.

Conclusions

Software metrics have been successfully employed for decades to understand, monitor, and improve products and processes. There are volumes of literature describing successes and methods, and organizations regularly implement metrics programs. In software development and quality assurance there is almost blind acceptance of the value of such programs, even though in many of these same organizations the metrics program is secretly causing lower productivity and quality.

It is imperative for any organization interested in quality to be alert and careful about metrics. Even organizations that have well established programs, especially organizations with long established metrics programs, ought to consider whether the metrics have the desired meanings and identify what side effects are caused. Where efforts are diverted without improving the product or its quality, some questioning should be made as to the appropriateness of the measures and metrics. The unintended side effects may be slowing rather than streamlining the organization, and can even serve to obscure our understanding of test results and reduce the overall product quality.

Cem Kaner has provided a framework for understanding and rethinking software metrics, but observations of behaviors within organizations is often sufficient to recognize unintended side effects. By reassessing the meanings of our metrics and recognizing their limitations we can potentially reduce the negative impacts.