Creating and Using Quality Software Delivery Measurements and Metrics

Peter Caron

Introduction

In the 1990s, physicist turned management guru Eli Goldratt mused "if you tell me what you are going to measure, I'll tell you how I am going to behave."[1] In other words, measuring something influences its behaviour: whether the target is a photon or a software developer. Today, software development teams suffer from both inaccurate measurements of their performance and unrealistic expectations of what they can achieve in a given time. Both these shortcomings stem from the way many companies measure how their products are developed and delivered. In order to generate valuable metrics to accurately assess the quality and efficiency of software products and development practices, we need to keep Goldratt's supposition in mind and we should choose our metrics accordingly.

The discussion of how to measure developers is not at all new. Managers have tried in many ways to measure the quality of development work and even developers with varying degrees of success.

In this article I address the question of how to measure **developer workforce productivity** in general and **development effectiveness** specifically. As most seasoned managers will already have learned, development measurements are best assessed against groups of engineers aligned to an outcome: a service or a product delivers a tangible result or value, rather than against any individual activity: how hard a developer works.

Quality metrics, by which I mean here the quality of the metrics rather than of the final product, are a set of measurements employed over time to analyse the results of the efforts of a development team and to measure how well they have achieved the goals assigned to them (i.e. they have met the definition of done). It is an important element of an effective set of metrics that they are *outcome-oriented* as opposed to *output* or *activity-oriented* and that when possible, they provide *leading* rather than *lagging* indicators. This means that metrics should help avoid problems rather than simply presenting them after the release. In other words, the purpose of any useful software metric is to provide indictors that can be used to assess productivity, costs, quality, degree of automation, etc. while that work is in progress.

How effective development is achieved by any group is dependent upon many factors. Since developers rarely work in isolation they are often impacted by actions and decisions taken outside their own spheres of influence. There are also many business factors outside of a development team's control which hinder their daily tasks and work. An obvious example is a set of poorly described requirements which result in wasted time and effort for developers. Though this wasted time represents a clear burden on a company, there is, for managers, HR, Finance and others, no easy, objective way to measure an individual programmer's productivity or determine which individuals are doing better or faster work than others. It is also extremely difficult and risky to try to compare productivity across teams.

[1] Goldratt, E.M., The Haystack Syndrome, 1990.

Measurements

The Good, the Bad and the Inefficient

How then do we quantify development quality and effectiveness? It should come as no surprise that traditional metrics fail to fully satisfy development objectives, yet people still insist on proposing them as viable alternatives. Can't we measure bug escape levels or Defect Density? How about failing test to judge the quality of software? The assumption of the latter is especially insidious and is often found in companies that expect a "bug-free" release to the customer. These folks would do well to remember that tests are there to *fail* not to *pass*. Finding bugs in our system is the whole raison d'être of Continuous Integration.

Continuous Integration doesn't get rid of bugs, but it does make them dramatically easier to find and remove."

- Martin Fowler, Chief Scientist, ThoughtWorks

Yet, throughout the industry people continue to insist on tracking that either tell us little or are entirely misleading. Here are some examples of useless metrics:

- Lines of code (LOC) has an obvious flaw in that someone can write 20K lines of bad code in place of 1k of functional code to do the same thing.
- Defect Density (Defect/kLoC) usually calculated as defects per thousand lines of code is, like LOC above, fundamentally flawed. Redefining defects (by severity or timing) whether intentional or not alters the perception of quality. There are many problems with this kind of calculation: different languages generate different size code – sometime wildly so, do you calculate only new code and new bugs or all code and new bugs, ... But the biggest shortcoming of Defect Density is this: when a team is actively refactoring code by removing unneeded functions or rewriting poor classes, they will usually see *decreases* in the LoC. In such a case, their Defect Density will actually *increase* even if they produce the same or fewer defects in a certain period.
- Function points describe activity whereas we want to assess development teams on results. Not only do poor programmers require more function points to accomplish the same tasks as experts, but experts spend at least some of their time in activities which are not captured by points - thinking, coaching, code reviews, etc.

CI/CD guru and author, Jez Humble has written: "Our most productive people are those that find ingenious ways to avoid writing any code at all." Hence, you should avoid any LoC-based metric and avoid function points. Both are activity-based indicators as opposed to outcome-based.

Here are some other metrics that are easily abused or manipulated and should be avoided:

- Velocity: This often leads to point inflation
- Cycle time: This sometimes leads to people staying busy but not delivering any real value. Nevertheless, cycle time can be tracked and used as a positive indicator if done in combination with other measurements.

There are many more ways in which metrics can be manipulated to distort their usefulness, but we need to minimise these risks by focusing on useful calculations. Metrics can and will be interpreted to fit a narrative: often enough not of your own

choosing. Facts too can and will be twisted to fit someone's narrative at least as often as the other way around. This does not always imply a duplicitous action on anyone's part as historian Jerry Z. Muller describes in his enlightening book, The Tyranny of Metrics. Though not written with software developers in mind, the lessons in his book are equally applicable to the SDLC. If developers know what you are measuring and there is a way to manipulate the results, the results will change: intentionally or not.

Before we choose which metrics we want to collect, we first need to decide what we need these metrics to tell us. In other words, decide on which metrics are most relevant to describe our business objectives. Mapping metrics to business objectives is often one of the most challenging tasks for companies.

Failure Demand

Viable and supportable software development needs to avoid something called *failure demand*. Failure demand is work created by failing to do the right thing the first time. A measure that is reasonably indicative of development efficiency is *the amount of time spent on unplanned work and excess rework.* This is one of the biggest forms of waste in an organization. Unplanned work and rework leads to longer development and release cycles but also diverts resources from work that would otherwise add real value. If it is not already understood by everyone, it should be: **bugs and mistakes in software add significantly to the total cost of a project or product development**. Development rework costs are reflected in higher maintenance and support costs and there are also direct costs associated with the business: Downtime, security breaches, lost IP, lost customers, fines or lawsuits. Yet time and again companies either misrepresent the total cost of a project by ignoring the cost of rework in both estimates and final calculations.

Since 2014, Puppet Labs has published the "State of DevOps Report" in which they try to identify the key capabilities that drive software delivery performance. The authors have done their best to correlate what practices lead to high performing companies defined as those that release quality software fast.

Software metrics can be divided into at least two categories: outcome and input. As stated previously, *outcome-oriented* means measuring a state after software is released as opposed to *output* or *activity-oriented* which measures the effort used to achieve that release. To illustrate the difference an outcome metric is delivery frequency: we delivered a tested, working version of the software four times this month. An activity metric is person days required to create a release: we spent 440 person-days this month to ship a version to the customer including meetings, testing, etc. We want to concentrate on measuring outcome which represents to us a business value (i.e. we delivered something to the customer).

Outcome Metrics

One of the most effective ways to measure transformational change is to use metrics which are not easily manipulated, and which track outcomes – velocity in the form of throughput and quality by measuring service or product stability. The first two measurements (1,2) gauge throughput, the second two measurements gauge stability

(3,4) which in this case is a surrogate for stability and quality. Outcome metrics which best measure the **throughput and stability** of services[2] are:

1. **Delivery Frequency.** How often is a service deployed to production?
2. **Cycle time** or **Lead time for changes.** How much time elapsed from a change being decided or developed to getting into production?

3. **Mean Time to Recover (MTTR).** How long did it take to get a fix for a known issue into production? We can extend this by breaking it down into components
4. **Change Failure Rate (CFR).** Measure how many times deployed software needs to be fixed once in production.

Outcome metrics which best measure the **throughput and stability** of products[3] are:

1. **Delivery Frequency.** How often is a product ready to be shipped to the customer?
2. **Cycle time** or **Lead time for changes.** How much time elapsed from a change being decided or developed to getting into production?

3. **Mean Time to Reshipment (MTTR).** How long did it take to get a fix for a known issue to the customer?
4. **Change Failure Rate (CFR).** Measure how many times shipped software needs to be fixed once delivered.

As a trend measurement only, it is possible to define the above in a single number tracked over time. An E_{index} can be calculated from these measurements with a target for an individual or collection of services.

$$\frac{deliveries}{failures} \times \frac{LTopt \times MTTRopt}{LT \times MTTR} = Eindex$$

In this example, an E_{index} score which is higher is better, one that is lower is worse[4].

Input Metrics

To help assess the quality of the software itself, we can measure other development or input metrics explicitly associated with code and which can measure productivity for services development.
These include:

- % of new code covered by integration tests (should be >95%)
- Code and author churn / Code toxicity (see below)
    - Code churn and Author churn are measurements which help to identify potential hotspots in code. An index would be based on *complexity trends*. A measure can have "red" warnings during a given time frame.
- Code complexity, hotspots and toxicity.

Complexity and hotspots are well-defined and can be visualised using a tool like CodeScene from Empear. In CodeScene, hotspots are calculated by measuring the changes to a file over time and corelating them with a McCabe complexity across the

[2] "State of DevOps Reports", Puppet Labs, 2016.
[3] Ibid.
[4] Optimal (opt) are our target values.

same time. This allows development teams to identify and address the highest risk and most volatile code and hopefully insert it into the backlog. In this way, refactoring is an ongoing task.

One can also plot a trend over time as illustrated in the figure below:

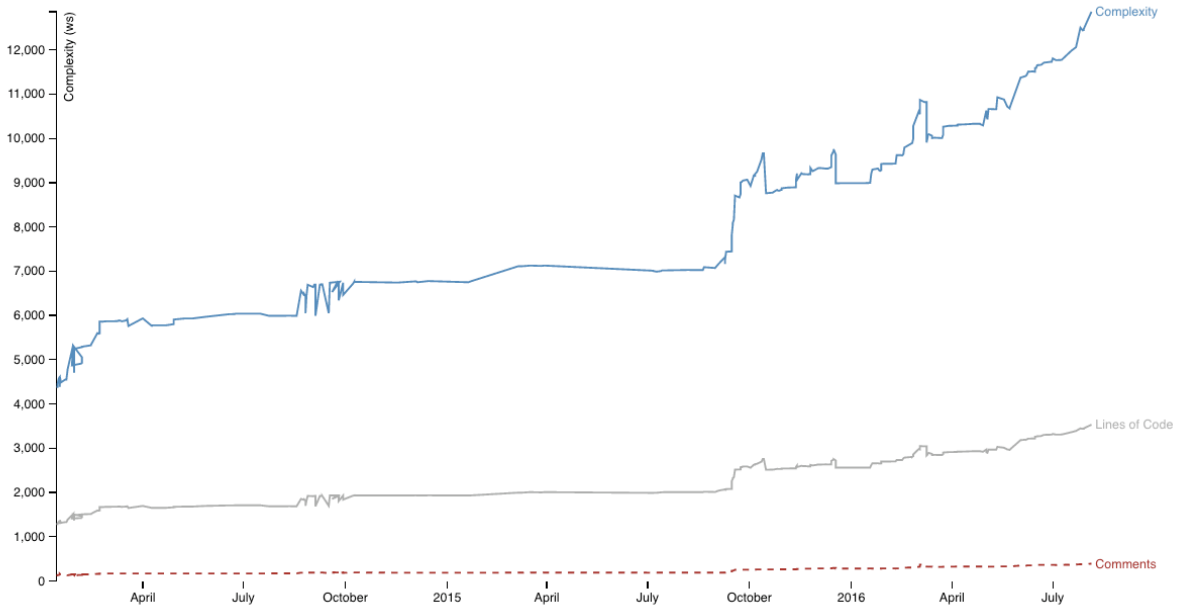

Figure 1 This is a complexity graph for a file showing the relationship between lines of code and complexity.

The above suggests that the complexity of this code has begun in October to grow rapidly. It is also growing non-linearly as the amount of new code increases. This may indicate that the code will become harder and harder to understand.

Again, a hotspot is *complicated code that you have to work with often.*[5] Hotspots are identified internally in development groups graphically or as a list.

[5] http://codescene.io/docs/guides/technical/hotspots.html

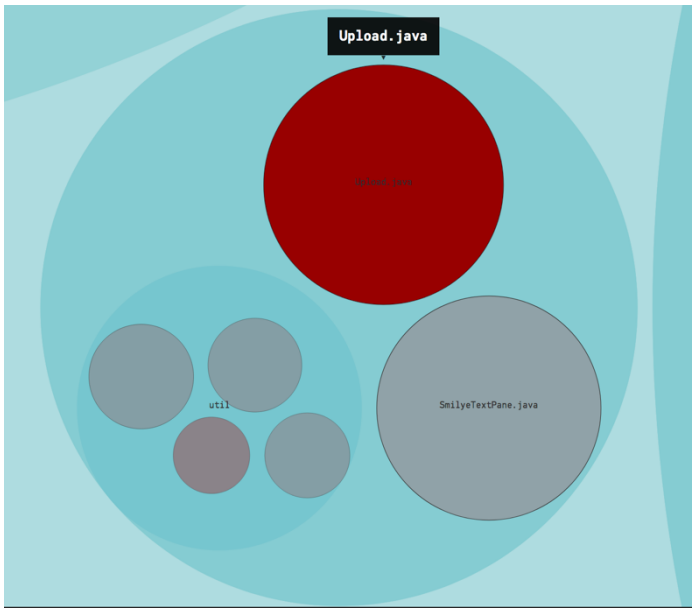

Figure 2 A graphical representation of code complexity based on code and author churn

Biomarkers

Another tool offered by CodeScene which provides a useful input metric is Code Biomarkers. According to the CodeScene authors, Code Biomarkers fill a number of important gaps in code analysis:

- *Bridge the gap between developers and non-technical stakeholders*: The biomarkers visualization provides information to managers that help decide on when to take a step back, invest in technical improvements, and measure the effects.
- *Get immediate feedback on improvements*: The biomarker trends gives you immediate and visual feedback on the investments you make in refactoring.
- *Share an objective picture of your code quality*: The biomarker scores are based on baseline data from thousands of codebases, and your code is scored against an industry average of similar codebases.
- *Get suggestions on where to start to refactoring*: The code biomarkers hint at specific problems in each file, which also suggests which refactoring that could be used to address the findings.

Code Biomarkers

Code Biomarkers aim to indicate specific properties of the code.

File	Status Now	Last Month	Last Year	Details
DBManager.java secureshares/web/src/main/java/ro/panzo/secureshares/db/	C	C	C	Low Overall Code Complexity High Degree of Code Duplication Primitive obsession Heavy usage of string arguments
Util.java secureshares/web/src/main/java/ro/panzo/secureshares/util/	B	B	B	Large Brain Method Detected Excess function arguments
secureshares.js secureshares/web/src/main/webapp/js/	A	A	A	Low Overall Code Complexity
InsertFileService.java secureshares/web/src/main/java/ro/panzo/secureshares/servlet/services/	A	A	A	OK
DownloadServlet.java secureshares/web/src/main/java/ro/panzo/secureshares/servlet/	A	B	B	OK
InsertUserService.java secureshares/web/src/main/java/ro/panzo/secureshares/servlet/services/	B	B	B	Many Conditionals Large Brain Method Detected

Figure 3 A Biomarker example

Toxicity

Toxicity is a measure of code with poor internal quality and is hard to maintain or extend. Toxicity in code can be index based on aggregated measures of common problems.[6]

Metric	Level	Threshold
File Length	file	500
Class Fan-Out Complexity	class	30
Class Data Abstraction Coupling	class	10
Anon Inner Length	inner class	35
Method Length	method	30
Parameter Number	method	6
Cyclomatic Complexity	method	10
Nested If Depth	statement	3
Nested Try Depth	statement	2
Boolean Expression Complexity	statement	3
Missing Switch Default	statement	1

[6] http://erik.doernenburg.com/2008/11/how-toxic-is-your-code/

Figure 4 The above table is an example of how toxicity might be measured. It shows metrics that make up a toxicity score and some example base thresholds on which the multipliers are based.[7]

Note: These measurements should be only applied to teams, services or programs. They cannot be used to measure individuals because of statistical variations in R&D teams[8].

Services

A service is defined here as anysoftware using application programming interfaces (APIs) to allow internal or external connections, partners or customers access to data. It is assumed that services are developed, managed and deployed in an automated way according to continuous delivery guidelines. Software development should be built on continuous integration and DevOps principles.[9] Services developed using these continuous integration methods should be continuously evaluated and continuously improved according to a standard set of outcome metrics. These metrics measure first and foremost the customer experience (feature expectations, stability and quality) but also includes speed of feature delivery, release frequency and ultimately costs.

Products

A product is here defined as the binary artefact of software development which is shipped to a customer (as embedded code, via an online store or as compiled code which once installed on a client cannot or should not be updated at the will of the developer. i.e. a library). For a client product (an SDK, an embedded app, internally developed tools and other client software that is delivered either to a customer or third-party integrator, we need a measurement which is a simple index calculation. There is a way to roughly calculate excess rework. Technical staff size includes all technical member of the extended development, release and operations team needed to deliver a feature set or full product to a customer.

Technical staff size x percentage of excess rework

Calculating excess rework can be done using JIRA tickets (i.e. “Show-stopper”, “Severe” and “Serious” tickets for example). % of excess rework *eRw* can be calculated

$$\mathrm{eRw} = \frac{Ft}{Bt}$$

where *Ft* represents *# of feature tickets* in a program and *Bt* is *# of P0 to P2 bug tickets after release*. For a more precise measure of excess rework the *delta* for time to complete any *Bt* tickets can be factored into the above equation.

Excess rework can also be used as a measure for services, but I have presented it as a product metric because the impact is higher in products than in services. Because products are often delivered less frequently (sometimes because of “middlemen” steps

[7] Ibid

[8] John Seddon, an occupational psychologist, researcher and professor found” It is worth bearing that in mind when deciding on which measurements to highlight in an LT environment.

[9] See various publications by Martin Fowler, Jez Humble, and others for more information on Continuous Integration, Continuous Delivery and DevOps

like iTunes or Google Play which delay releases to the end device or customer) the impact is compounded, whereas service APIs can be updated in smaller increments according to the principles of CD allowing fix forward solutions.

Alternatives

There are other ways to measure development efficiency and which can be used as trackable metrics for a workforce productivity assessment. Here are some examples for which care should be taken that they are not used in isolation.

- Green Builds / day: The number of times a release candidate (green build) is available from the continuous integration systems.
- Cycle time accurately and consistently measured from code commit to release into production
- Percentage of promised Features delivered according to JIRA tickets (possibly using similarly sized tickets)
- Code toxicity / churn: as above
- Percentage of new code covered by unit tests

Transparency via Dashboards

Below are some common metrics that should be applied across development teams regardless of the product or service they deliver. For each Business Group that produces different products and services outcomes are sometimes and partially automated, complementary and occasionally duplicated efforts in the form of shadow IT, CI, QA and so-called "*integration*" and even antithetical DevOps teams. The development processes across these business units should share commonalities (standard ways of working) and used shared infrastructures and services where appropriate and from which it is possible to better assess workforce productivity in a standard manner.

Code Review

A code review process is a proven method to finding defects in software when done consistently and properly. Some code review techniques are more efficient and effective than others. Generally, the more defects found, the better the process is working (see next section). This page draws heavily on "Best Kept Secrets of Peer Code Review" by Jason Cohen, Steven Teleki and Eric Brown by SmartBear Software and based on research conducted at and in cooperation with Cisco Systems.

Description

The inclusion of a common code review process in development is the first step in the overall improvement of quality and time-to-delivery. A successful implementation of a peer review process requires that management (or the leaders of the development groups) foster a proper code review culture. Finding defects early needs to be understood as a positive thing - a good thing. Lightweight style code reviews can be efficient and effective at finding bugs but review processes and review metrics cannot be used to single out developers!

The cost of fixing defects increases exponentially as software progresses through the development lifecycle, therefore it's critical to catch defects as early as possible. A study from IBM showed the following:

Design and architecture	Implementation	Integration testing	Customer beta test	Postproduct release
1X*	5X	10X	15X	30X

*X is a normalized unit of cost and can be expressed in terms of person-hours, dollars, etc.
Source: National Institute of Standards and Technology (NIST)†

By catching defects as early as possible in the development cycle, you can significantly reduce your development costs.

Some things to keep in mind when implementing a peer code review process:

1. Difficult or Complex Code has more defects
2. More time yields more defects (up to the limit which a reviewer can concentrate in a single session)
3. The objective is to produce better code
4. The more defects the better

Suggested Practices

1. Review only 100 to 300 lines of code per session. A Cisco study has suggested that as the number of lines of code under review grows beyond 300, defect density drops off considerably.
2. Inspection rates less than 400 LoC/hr
3. Limit the time per session to 60-90 minutes
4. Require proper annotation of source code prior to review
5. Establish quantifiable goals for code review and capture those metrics

    - External metrics, such as "reduce support calls by 20%" or "halve the percentage of defects injected by development." This information gives you a clear picture of how your code is doing from the outside perspective, and it needs to be a quantifiable measurement, not just a vague goal to "fix more bugs."
    - Consider that only automated or tightly controlled processes can give you repeatable metrics; humans aren't good at remembering to stop and start stopwatches. For best results, use a code review tool that gathers metrics *automatically* so that your critical metrics for process improvement are accurate.
    - It's also useful to watch internal process metrics to get an idea of how many defects are found, where your problems lie, and how long your developers are spending on reviews. The most common internal metrics for code review are *inspection rate*, *defect rate*, and *defect density*.

1. Create and use checklists to improve results for developers and reviewers see checklist below
2. Verify that defects are really fixed
3. Log all defects and decisions

    - Defects are logged like comments, also threaded by file and line number. When an author believed a defect had been fixed, the new files were uploaded to the same review.
    - Once all reviewers agree the review is complete and no defects are still open, the review is complete and the developer is then allowed to commit.

Adapted from 11 Best Practices of Peer Code Review[1]

Things to Avoid

1. Reviews of enormous amounts of code. If many thousands of lines of code were under review, we can be sure this is not a true code review.
2. Trivial reviews. These are reviews in which clearly the reviewer never looked at the code, or at least not long enough for any real effect. For example, if the entire review took two seconds, clearly no review took place.
3. Using code review to evaluate developers
4. Using code reviews as a measure of development speed

Metrics

Metric	Description
Inspection Rate	How fast are we able to review code? Normally measured in kLOC (thousand Lines of Code) per man-hour.
Defect Rate	How fast are we able to find defects? Normally measured in number of defects found per man-hour.
Defect Density	How many defects do we find in a given amount of code (not how many there are)? Normally measured in number of defects found per kLOC.

Conclusion

No matter how we measure efficiency or productivity in software development, we must be mindful to focus our attentions on **measuring outcomes: those products and services which can be measured according to the value of strategic objectives they produce**. Once we begin to accurately measure our work, we can then **identify and eliminate wasted time**. The way we calculate inefficiencies will vary between services and products. For a service, factors like regular deployments to production and limited time to recover from a bug or failed deployment are key indicators of efficiency. For a product was can assess the results of full-verification testing and draw inferences about quality from that. How we use (or abuse) the statistics and information we collect is the subject of the next chapter, but we should try to keep in mind something Rudyard Kipling

wrote quoting Mark Twain: “Get your facts first, and then you can distort 'em as much as you please.”[10]

The objective of this chapter is to propose that we direct our focus to understanding and measuring software development *outcomes* instead of simply monitoring *activity* or *output*. Therefore, our metrics must reflect this. The most advantageous and simple outcome is *a product or service delivered rapidly, on-time and with high quality.*

[10] Rudyard Kipling, *From Sea to Shining Sea,*

Bibliography

Goldratt, E. (1990). *The Haystack Syndrome.* New York: North River Press.

Humble, J. J. (2015). *Lean Enterprise: How High Performance Organizations Innovate at Scale* . Sebastopol, CA: O'Reilly.

Morris, K. (2016). *Infrastructure as Code.* Sebastopol, CA: O'Reilly.

Nicolette, D. (2015). *Software Development Metrics.* Shelter Island, NY: Manning.

Tornhill, A. (2015). *Your Code as a Crime Scene.* Dallas, TX: The Pragmatic Programmers.

Doernenburg, Aaron, (2008) *How Toxic is your Code*, http://erik.doernenburg.com/2008/11/how-toxic-is-your-code/