acmqueue

# Getting What You Measure

**Four common pitfalls in using software metrics for project management**

Eric Bouwers, Software Improvement Group and Delft University of Technology
Joost Visser, Software Improvement Group and Radboud University Nijmegen
Arie van Deursen, Delft University of Technology

Software metrics—helpful tools or a waste of time? For every developer who treasures these mathematical abstractions of software systems there is a developer who thinks software metrics are invented just to keep project managers busy. Software metrics can be very powerful tools that help achieve your goals but it is important to use them correctly, as they also have the power to demotivate project teams and steer development in the wrong direction.

For the past 11 years, the Software Improvement Group has advised hundreds of organizations about software development and risk management on the basis of software metrics. We have used software metrics in more than 200 investigations in which we examined a single snapshot of a system. Additionally, we have used software metrics to track the ongoing development effort of more than 400 systems. In the course of these projects, we have learned some pitfalls to avoid when using software metrics for project management. This article addresses the four most important of these:

- Metric in a bubble
- Treating the metric
- One-track metric
- Metrics galore

Knowing about these pitfalls will help you to recognize them and, hopefully, avoid them, which will help make your project successful. As a software engineer, your knowledge of these pitfalls will help you understand why project managers want to use software metrics and assist the managers when they are applying metrics in an inefficient manner. As an outside consultant, you need to take the pitfalls into account when presenting advice and suggestions. Finally, if you are doing research in the area of software metrics, knowing these pitfalls will help you place your new metric in context when presenting it to practitioners. Before diving into the pitfalls, let's look at why software metrics can be useful.

## SOFTWARE METRICS STEER PEOPLE

*"You get what you measure."* This phrase definitely applies to software project teams. No matter what you define as a metric, as soon as it is used to evaluate a team, the value of the metric moves toward the desired value. Thus, to reach a goal, you can continuously measure properties of the desired goal and plot these measurements in a place visible to the team. Ideally, the desired goal is plotted alongside the current measurement to indicate the distance to the goal.

Imagine a project in which the runtime performance of a particular use case is of critical importance. In this case it helps to create a test in which the execution time of the use case is measured daily. By plotting this daily data point against the desired value, and making sure the team

sees this measurement, it becomes clear to everybody whether the target is being met or whether the recent development actions are leading the team away from the goal.

Even though it might seem simple, this technique can be applied incorrectly in a number of subtle ways. For example, imagine a situation in which customers are unhappy because they report problems in a product that are not solved in a timely manner. To improve customer satisfaction, the project team tracks the average resolution time for issues in a release, following the reasoning that a lower average resolution time increases customer satisfaction.

Unfortunately, reality is not so simple. First, solving issues faster might lead to unwanted side effects—for example, a quick fix now could result in longer fix times later because of incurred technical debt. Second, solving an issue within days does not help the customer if these fixes are released only once a year. Finally, customers are undoubtedly more satisfied when no fix is required at all—that is, issues do not end up in the product in the first place.

Thus, using a metric allows you to steer toward a goal, which can be either a high-level business proposition ("the costs of maintaining this system should not exceed $100,000 per year") or more technically oriented ("all pages should load within 10 seconds"). Unfortunately, using metrics can also prevent you from reaching the desired goal, depending on the pitfalls encountered. In the remainder of this article we are going to discuss some of the pitfalls we frequently encountered and explain how they can be recognized and avoided.

## WHAT DOES THE METRIC MEAN?

Software metrics can be measured on different views of a software system. This article focuses on metrics calculated on a particular version of the code base of a system, but the pitfalls also apply to metrics calculated on other views.

Assuming that the code base contains only the code of the current project, software product metrics establish a ground truth. Calculating only the metrics is not enough, however. Two more actions are needed to interpret the value of the metric: adding *context*;, and establishing the relationship with the *goal*.

To illustrate these points, we use the LOC (lines of code) metric to provide details about the current size of a project. Even though there are multiple definitions of what constitutes a line of code, such a metric can be used to reason about whether the examined code base is complete or contains extraneous code such as copied-in libraries. To do this, however, the metric should be placed in context, bringing us to our first pitfall.

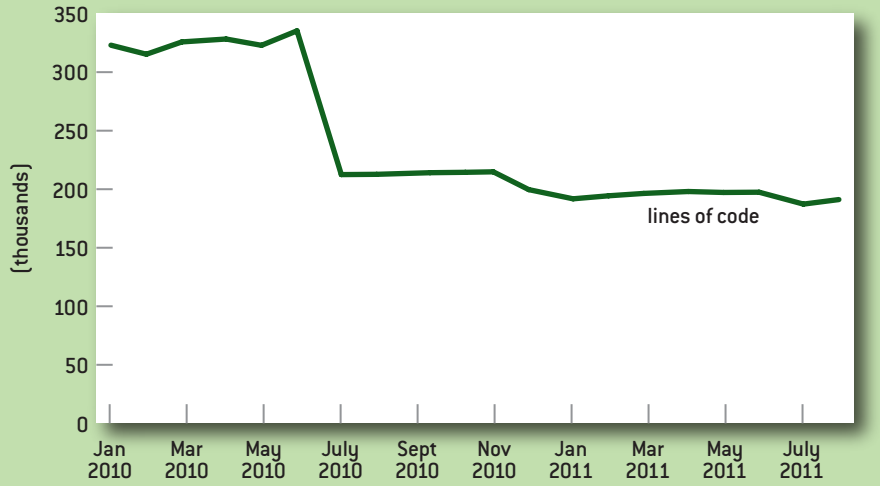
### METRIC IN A BUBBLE

*Using a metric without proper interpretation. Recognized by not being able to explain what a given value of a metric means. Can be solved by placing the metric in a context with respect to a goal.*

The usefulness of a single data point of a metric is limited. Knowing that a system is 100,000 LOC is meaningless by itself, since the number alone does not explain if the system is large or small. To be useful, the value of the metric should, for example, be compared against data points taken from the history of the project or from a benchmark of other projects. In the first scenario, you can discover trends that should be explained by external events. For example, the graph in figure 1 shows the LOC of a software system from January 2010 to July 2011.

FIGURE 1

**Lines of Code of a Software System**




The first question that comes to mind here is: "Why did the size of the system drop so much in July 2010?" If the answer to this question is, "We removed a lot of open source code we copied in earlier," then there is no problem (other than the inclusion of this code in the first place). If the answer is, "We accidentally deleted part of our code base," then it might be wise to introduce a different method of source-code version management. In this case the answer is that an action was scheduled to drastically reduce the amount of configuration needed; given the amount of code that was removed, this action was apparently successful.

Note that one of the benefits of placing metrics in context is that it allows you to focus on the important part of the graph. Questions about what happened at a certain point in time or why the value significantly deviates from other systems become more important than the specific details about how the metric is measured. Often people, either on purpose or by accident, try to steer a discussion toward "How is this metric measured?" instead of "What do these data points tell me?" In most cases the exact construction of a metric is *not important for the conclusion drawn from the data*. For example, consider the three plots shown in figures 2 and 3 representing different ways of computing the *volume* of a system. Figure 2 shows the lines of code counted as every line containing at least one character that is not a comment or white space (orange) and lines of code counted as all new line characters (blue). Figure 3 shows the number of files used.

The trend lines indicate that, even though the scale differs, these volume metrics all show the same events. This means that each of these metrics is a good candidate to compare the volume of a system against other systems. As long as the volume of the other systems is measured in the same manner, the conclusions drawn from the data will be very similar.

The different trend lines bring up a second question: "Why does the volume decrease after a period in which the volume increased?" The answer can be found in the normal way in which alterations are made to this particular system. When the volume of the system increases, an action is

FIGURE 2

**Measuring Lines of Code in Two Different Ways**

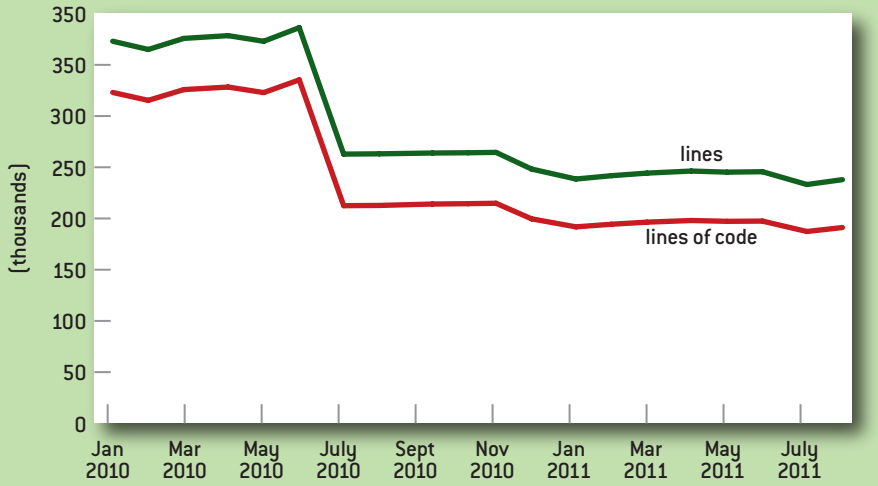
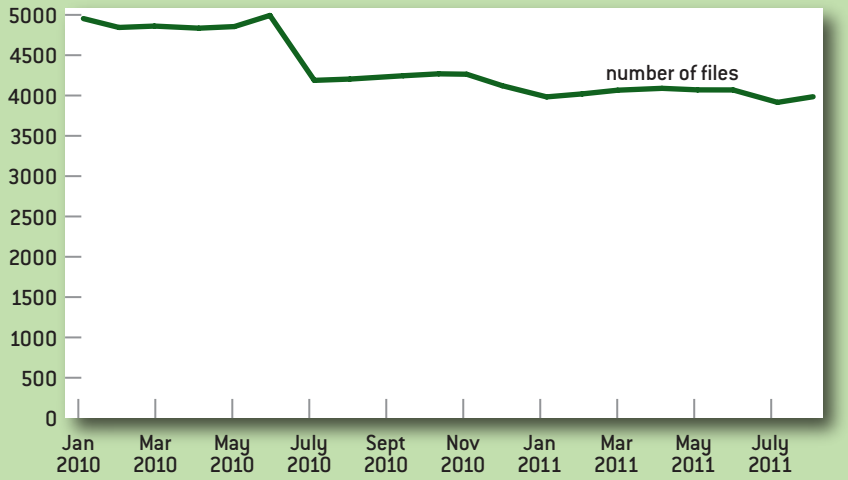



FIGURE 3

**Measuring the Number of Files Used**




scheduled to determine whether new abstractions are possible, which is usually the case. This type of refactoring can significantly decrease the size of the code base, which results in lower maintenance effort and makes it easier to add functionality to the system. Thus, the goal here is to reduce maintenance effort by (among others) keeping the code base relatively small.

In the ideal situation a direct relationship exists between a desired goal (e.g., reduced maintenance effort) and a metric (e.g., a small code base). In some cases this relationship is based on informal

reasoning (e.g., when the code base of a system is small it is easier to analyze what the system does); in other cases scientific research has shown that the relationship exists. What is important here is that you determine both the nature of the relationship between the metric and the goal (direct/indirect) and the strength of this relationship (informal reasoning/empirically validated).

Thus, a metric in isolation will not help you reach your goal. On the other hand, assigning too much meaning to a metric leads to a different pitfall.

#### TREATING THE METRIC

*Making alterations just to improve the value of a metric. Recognized when changes made to the software are purely cosmetic. Can be solved by determining the root cause of the value of a metric.*

The most common pitfall is making changes to a system just to improve the value of a metric, instead of trying to reach a particular goal. At this point, the value of the metric has become a goal in itself, instead of a means of reaching a larger goal. This situation leads to refactorings that simply "please the metric," which is a waste of precious resources. You know this has happened when, for example, one developer explains to another developer that a refactoring needs to be done because "the duplication percentage is too high," instead of explaining that multiple copies of a piece of code can cause problems in maintaining the code. It is never a problem that the value of a metric is too high or too low: the fact that this value is not in line with your goal should be the reason to perform a refactoring.

Consider a project in which the number of parameters for methods is high compared with a benchmark. When a method has a relatively large number of parameters (e.g., more than seven) it can indicate that this method is implementing different functionalities. Splitting the method into smaller methods would make it easier to understand each function separately.

A second problem that could be surfacing through this metric is the lack of a grouping of related data objects. For example, consider a method that takes as parameters a Date object called `startDate` and another called `endDate`. The names suggest that these two parameters together form a `DatePeriod` object in which `startDate` will need to be before `endDate`. When multiple methods take these two parameters as input, introducing such a `DatePeriod` object to make this explicit in the model could be beneficial, reducing both future maintenance effort and the number of parameters being passed to methods.

Sometimes, however, parameters are, for example, moved to the fields of the surrounding class or replaced by a map in which a `(String,Object)` pair represents the different parameters. Although both strategies reduce the number of parameters inside methods, if the goal is to improve readability and reduce future maintenance effort, then these changes are not helping. It could be that this type of refactoring is done because the developers simply do not understand the goal and thus are treating the symptoms. There are also situations, however, in which these non-goal-oriented refactorings are done to game the system. In both situations it is important to make the developers aware of the underlying goals to ensure that effort is spent wisely.

Thus a metric should never be used as is, but it should be placed inside a context that enables a meaningful comparison. Additionally, the relationship between the metric and the desired property of the system should be clear; this enables you to use the metric to schedule specific actions that will help reach your goal. Make sure that the scheduled actions are targeted toward reaching the underlying goal instead of only improving the value of the metric.

## HOW MANY METRICS DO YOU NEED?

Each metric provides a specific viewpoint on your system. Therefore, combining multiple metrics leads to a balanced overview of the current state of your system. The number of metrics used can lead to two pitfalls. First, consider using only a single metric.

### ONE-TRACK METRIC

*Focusing on only a single metric. Recognized by seeing only one metric (of just a few) on display. Can be solved by adding metrics relevant to the goal.*

Using only a single software metric to measure whether you are on track toward your goal reduces that goal to a single dimension (i.e., the metric that is currently being measured). A goal is never one-dimensional, however. Software projects experience constant tradeoffs between delivering desired functionality and nonfunctional requirements such as security, performance, scalability, and maintainability. Therefore, multiple metrics are necessary to ensure that your goal, including specified tradeoffs, is reached. For example, a small code base might be easier to analyze, but if this code base is made of highly complex code, then it can still be hard to make changes.

In addition to providing a more balanced view of your goal, using multiple metrics also assists you in finding the root cause of a problem. A single metric usually shows only a single symptom, while a combination of metrics can help diagnose the actual disease within a project.

For example, in one project the `equals` and `hashCode` methods (those used to implement equality for objects in Java) were among the longest and most complex methods within the system. Additionally, a relatively large percentage of duplication occurred in these methods. Since they use all the fields of a class, the metrics indicate that multiple classes have a relatively large number of fields that are also duplicated. Based on this observation, we reasoned that the duplicated fields formed an object that was missing from the model. In this case we advised looking into the model of the system to determine whether extending the model with a new object would be beneficial.

In this example, examining the metrics in isolation would not have led to this conclusion, but by combining several unit-level metrics, we were able to detect a design flaw.

### METRICS GALORE

*Focusing on too many metrics. Recognized when the team ignores all metrics. Can be solved by reducing the number of metrics used.*

Although using a single metric oversimplifies the goal, using too many metrics makes it hard (or even impossible) to reach your goal. Not only is it hard to find the right balance among a large set of metrics, it is bad for morale when a team sees that every change they make results in the decline of at least one metric. Also, when the value of a metric is far from the goal, then a team can start to think, "We will never get there, anyway," and simply ignore the metrics altogether.

For example, there have been multiple projects that deployed a static-analysis tool without critically examining the default configuration. When the tool in question contains, for example, a check that flags the use of a tab character instead of spaces, the first run of the tool can report an enormous number of violations for each check (running into the hundreds of thousands). Without proper interpretation of this number, it is easy to conclude that zero violations cannot be reached

within any reasonable amount of time (even though some problems can easily be solved by a simple formatting action). Such an incorrect assessment sometimes results in the team considering the tool useless and deciding to ignore it.

Fortunately, in other cases the team adapts the configuration to suit the specific situation by limiting the number of checks (e.g., by removing checks that measure highly related properties, can be solved automatically, or are not related to the current goals) and instantiating proper default values. Using such a specific configuration, the tool reports a lower number of violations that can be fixed in a reasonable amount of time.

To ensure that all violations are fixed eventually, the configuration can be extended to include other types of checks or stricter versions of checks. This will increase the total number of violations found, but when done correctly the number of reported violations does not demotivate the developers too much. This process can be repeated to extend the set of checks slowly toward all desired checks without overwhelming the developers with a large number of violations at once.

## CONCLUSION

Software metrics are useful tools for project managers and developers alike. To benefit from the full potential of metrics, keep the following recommendations in mind:

- Attach meaning to each metric by placing it in context and defining the relationship between the metric and your goal, and avoid making the metric a goal in itself.
- Use multiple metrics to track different dimensions of your goal, but avoid demotivating a team by using too many metrics.

If you are already using metrics in your daily work, try to link them to specific goals. If you are not using any metrics at this time but would like to see their effects, we suggest you start small: define a small goal (methods should be simple to understand for new personnel); define a small set of metrics (e.g., length and complexity of methods); define a target measurement (at least 90 percent of the code should be simple); and install a tool that can measure the metric. Communicate both the goal and the trend of the metric to your colleagues and experience the influence of metrics.

## LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

**ERIC BOUWERS** is a software engineer and technical consultant at the Software Improvement Group in Amsterdam, The Netherlands. He is a part-time Ph.D. student at Delft University of Technology. He is interested in how software metrics can assist in quantifying the architectural aspects of software quality. He can be reached at e.bouwers@sig.eu.

**JOOST VISSER** is head of research at the Software Improvement Group in Amsterdam, The Netherlands, where he is responsible for innovation of tools and services, academic relations, internship coordination, and general research. He also holds a part-time position as professor of large-scale software systems at the Radboud University Nijmegen, The Netherlands. He can be reached at j.visser@sig.eu.

**ARIE VAN DEURSEN** is a full professor in software engineering at Delft University of Technology, The Netherlands, where he leads the Software Engineering Research Group. His research topics include software testing, software architecture, and collaborative software development. He can be reached at Arie.vanDeursen@tudelft.nl.

© 2012 ACM 1542-7730/12/0500 $10.00