Getting what you measure: four common pitfalls in using software metrics for project management

Eric Bouwers[1,2], Joost Visser[1,3], and Arie van Deursen[2]

[1]Software Improvement Group
{e.bouwers,j.visser}@sig.eu
[2]Delft Technical University
Arie.vanDeursen@tudelft.nl
[3]Radboud University Nijmegen

Software metrics, a helpful tool or a waste of time? For every developer who treasures these mathematical abstractions of their software system there is a developer who thinks software metrics are only invented to keep their project managers busy. Software metrics can be a very powerful tool which can help you in achieving your goals. However, as with any tool, it is important to use them correctly, as they also have the power to demotivate project teams and steer development into the wrong direction.

In the past 11 years, the Software Improvement Group has been using software metrics as a basis for their management consultancy activities to identify risks and steer development activities. We have used software metrics in over 200 investigations in which we examined a single snapshot of a system. Additionally, we use software metrics to track the ongoing development effort of over 400 systems. While executing these projects, we have learned some pitfalls to avoid when using software metrics in a project management setting. In this article we discuss the four most important ones:

- 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 ultimately leads to being able to make your project more successful. As a software engineer, knowing these pitfalls helps in understanding why project managers want to use software metrics and help you in assisting them when they are applying metrics in an inefficient manner. For an external advisor, the pitfalls need to be taken into account when presenting advice and proposing actions. Lastly, should you be doing research in the area of software metrics, then it is good to know these pitfalls in order to place your new metric in the right context when presenting them to practitioners. But before diving into the pitfalls we first discuss why software metrics can be considered a useful tool.

1 Software metrics steer people

"You get what you measure": in our experience this phrase definitely applies to software project teams. No matter what you define as a metric, as soon as the metric is being used to evaluate a team the value of the metric moves towards the desired value. Thus, to reach a particular goal one can continuously measure properties of the desired goal and plotting these measurements in a place visible to the team. Ideally, the desired goal is plotted alongside the current measurement to indicate the current distance to the desired goal.

Imagine a project in which the run-time performance of a particular use-case is of critical importance. It then helps to create a test in which the execution time of the use-case is measured on a daily basis. 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 desired target is being met, or whether the development actions of yesterday are leading the team away from the goal.

Even though it might seem simple, there are a number of subtle ways in which this technique can be applied incorrectly. For example, imagine a situation in which customers are unhappy because issues that are found in a product are reported, but not solved in a timely matter. In order to improve customer satisfaction, the project team is tracking the average resolution time for issues in a release, following the reasoning that a lower average resolution time results in higher customer satisfaction.

Unfortunately, reality is not as simple as this. To start, solving issues faster might lead to unwanted side-effects, for example because a quick fix now results in longer fix-times later on due to incurred technical debt. Secondly, solving an issue within days does not help the customer if these fixes are released only once a year. Lastly, customers are probably more satisfied when issues do not end up in the product at all instead of them being fixed rapidly.

Thus by choosing a metric it becomes possible to steer towards a goal, but it can also make you never reach the desired goal at all. In the remainder of this article we will go over some of the pitfalls that you want to avoid when using metrics to reach a particular goal. Such a goal can either be a high-level business goal ("the costs of maintaining this system should not exceed 100K per year") or more technically oriented goals ("all pages should load within 10 seconds").

2 What does the metric mean?

Software metrics can be measured on different views of a software system. In this article we focus 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.

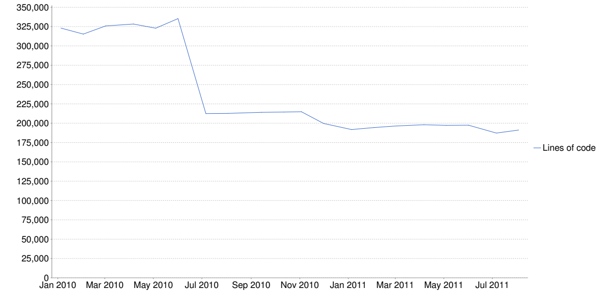
Assuming that the code-base only contains the code of the current project, software product metrics establish a ground-truth on which can be reasoned. However, only calculating the metrics is not enough. Two more actions are needed in order to interpret the value of the metric: *context* should be added and the relationship with the *goal* should be established.

To illustrate these points we use the 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' (LOC), 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. However, to do this the metric should be placed in a context, bringing us to our first pitfall.

2.1 Metric in a bubble

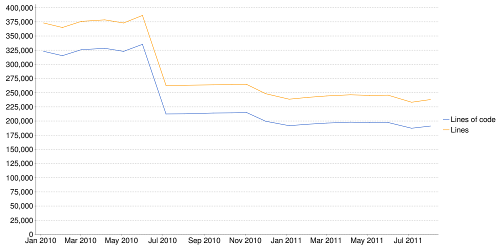
Using a metric without proper interpretation. Recognized by not being able to explain what a given value of a metric means. Can be solved by placing the metric inside a context with respect to a goal.

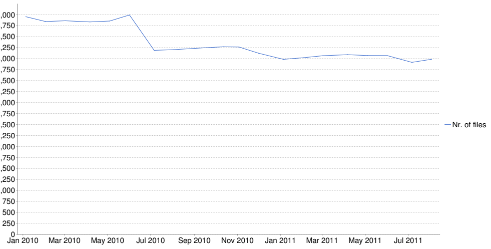
The usefulness of a single datapoint of a metric is limited. Knowing that a system is 100,000 LOC is meaningless by itself, since the number alone does not explain if the system is large or small. In order to be useful the value of the metric should, for example, be compared against data-points taken from the history of the project or taken from a benchmark of other projects. In the first scenario, trends can be discovered which should be explained by outside external events. For example, the following graph shows the LOC of a software system from January 2010 up until July 2011:



The first question that comes to mind here is: 'Why did the size of the system drop so much on July 2010?' If the answer to this question is 'we removed a lot of open-source code we copied-in earlier' there is no problem (other then the inclusion of this code in the first place). Should the answer be 'we accidentally deleted part of our code-base', then it might be wise to introduce a different way 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 which was removed this action was apparently successful.

Note that one of the benefits of placing metrics inside a context is that it enables you to focus on the important part of the graph. Questions regarding 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. We often encounter situations in which people, either on purpose or by accident, try to steer a discussion towards 'how is this metric measured' instead towards 'what do these datapoints tell me'. In most cases, the exact construction of a metric is *not important for* *the conclusion drawn from the data*. For example, below are three plots representing different ways of computing the *volume* of a system: (1) lines of code counted as every line containing at least one character which is not a comment or white-space; (2) lines of code counted as all new line characters; and (3) number of files used.





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

Looking at the different trend-lines a second question which could come to mind is: 'Why does the volume decrease after a period in which the volume increased?' The answer to this question 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 done to determine whether there are new abstractions 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 easier ways to add new functionality to the system. Thus the goal here is to reduce maintenance effort by (amongst others) keeping the size of the code-base relatively small.

In the ideal situation there is a direct relationship between a desired goal (i.e., reduced maintenance effort) and a metric (i.e., 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 been done to show 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) when determining the meaning of a metric.

Summarizing, 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.

2.2 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 we encounter is the situation in which changes are made 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 to reach a larger goal. This situation leads to refactorings to 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 for maintaining the code later on. 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.

To illustrate, we have encountered different projects in which the number of parameters for methods was relatively high (as compared to a benchmark). When a method has a relatively large number of parameters (e.g., more than seven), this can indicate that the method is implementing different functionalities. Splitting the method up into smaller methods would help in making it easier to understand each separate functionality.

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

However, we sometimes see that 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, it is clear that if the goal is to improve readability and reduce future maintenance effort these solutions are not helping you to reach your goal. 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, but we also encounter situations in which these non-goal-oriented refactorings are done to game the system on purpose. In both situations it is important to talk to the developers and make them aware of the underlying goals.

To summarize, a metric should never be used 'as-is', but placed inside a context which enables a meaningful comparison. Additionally, the relationship between the metric and desired property of your goal should be clear, this enables you to use the metric to schedule specific actions in order to reach your goal. However, make sure that scheduled actions are targeted towards reaching the underlying goal instead of only improving the value of the metric.

3 How many metrics do you need?

Each metric which is measured provides a specific view-point on your system. Therefore, combining multiple metrics allows you to get a well-balanced overview of the current state of your system. There are two pitfalls related to the number of metrics to be used. We start with using only a single metric.

3.1 One track metric

Focussing on only a single metric. Recognized by seeing only one (of just a few) metrics 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 towards your goal reduces your goal to only a single dimension, i.e., the metric which is currently being measured. However, a goal is never one-dimensional. For software projects there are constant trade-offs between delivering desired functionality and non-functional requirements such as security, performance, scalability and maintainability. Therefore, multiple metrics need to be used to ensure that your goal, including specified trade-offs, is reached. For example, a small code-base might be easier to analyze, but if this code-base is made of highly-complex code it is still be hard to make changes.

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

For example, in one project we found that the ’equals’ and ’hashCode’-methods (the methods used to implement equality for objects in Java) were amongst the longest and most complex methods within the system. Additionally, there was a relatively large percentage of duplication amongst these methods. Since these methods use all the fields of a class the metrics indicate that multiple classes have a relatively large number of fields which are also duplicated. Based on this observation we reasoned that the duplicated fields form an object that was missing from the model. In this case we advised to look 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 lead to this conclusion, but by combining several unit-level metrics we were able to detect a design flaw.

3.2 Metrics galore

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

Where using a single metric oversimplifies the goal, using too many metrics makes it hard (or even impossible) to reach your goal. Apart from making it hard to find the right balance amongst a large set of metrics, it is not motivating for a team to see that every change they make results in the decline of at least one metric. Additionally, when the value of a metric is far off the desired goal a team can start to think ’we will never get there anyway’ and simply ignore the metric all together.

For example, we have seen multiple projects in which a static analysis tool was deployed without critically examining the default configuration. When the tool in question contains, for example, a check that flags the usage of a tab-character instead of the use of spaces, the first run of the tool can report an enormous number of violations for each check (numbers running into the hundred of thousands). Without proper interpretation of this number it is easy to conclude that reaching zero violations cannot be done within any reasonable amount of time (even though some problems can easily be solved by a simple formatting action). Such an incorrect assessment sometimes leads to results in the tool being considered useless by the team, which then decides to ignore the tool.

Fortunately, in other cases the team adapts the configuration to suit their 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. By using such a specific configuration, the tool reports a lower number of violations which can be fixed in a reasonable amount of time.

In order to still ensure that all violations are fixed eventually, the configuration can be extended to include other types of checks or more strict 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 slowly extend the set of checks towards all desired checks, without overwhelming the developers with a large number of violations at once.

4 Conclusion

Software metrics are a useful tool which offer benefits for project managers and developers alike. In order to use the full potential of metrics, keep the following recommendations in mind:

- Attach meaning to each metric by placing it in context and by defining the relationship between the metric and your goal, while at the same time avoiding to make 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 see whether it is possible for you to link the metrics to specific goals. If you are not using any metrics at this time but like to see its 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 which can measure the metric. Communicate both the goal and the trend of the metric to your co-workers and see the influence of metrics at work.

5 Biographies

Eric Bouwers is a software engineer and technical consultant at the Software Improvement Group (SIG) in Amsterdam, The Netherlands. Additionally, he is a part-time PhD 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`.

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`.

Joost Visser is Head of Research at the Software Improvement Group (SIG) in Amsterdam, The Netherlands. In that role, Joost is responsible for innovation of tools and services, academic relations, internship coordination, and general research. Joost 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`.