**Title**: Metrics for Measuring the Quality of Object-Oriented Software

**Link**: <http://delivery.acm.org.libproxy.auc.ca/10.1145/2510000/2507311/p66b-singh.pdf?ip=199.212.55.169&id=2507311&acc=ACTIVE%20SERVICE&key=FD0067F557510FFB%2E2E114FAB5F912086%2E4D4702B0C3E38B35%2E4D4702B0C3E38B35&CFID=939957675&CFTOKEN=22411919&__acm__=1495551123_66186ebadebfcf7a06c0c6f881edc6c7>

**Summary:**

The paper presents five parameters for measuring software of object-oriented programming, and some associated metrics for each one. The article takes the approach that “anything which can’t be measured can’t be controlled”, and in part shows that changes to a piece of software overtimes can reduce the quality. The following metrics and parameters were used:

* LOC (lines of code): represents the size of the software, and states that the greater the size the less understandable it will be (measure of understandability)
* CC (cyclomatic complexity): measure the number of linearly independent paths through a program, and low values are considered better (measure of complexity)
* CBO (coupling between objects): metric for a class is the count of those classes which the given class is coupled, and higher values are seen as bad (measure of efficiency reuse, and complexity)
* LCOM (lack of cohesion): the degree to which methods within a class are related to one another, and maximized cohesion is seen as a positive (reuse, efficiency, complexity)
* WMC (weighted methods per class): indicator of how much time and effort is require to develop/maintain a class, and low is seen as greater polymorphism (measure of understandability, complexity, and reuse)
* RFC (response for a class): measure the number of methods that are invoked in response to a message received by an object of the class, and shows how much communication there is between classes which can complicate debugging/testing (measure of understanding and complexity)
* MI (maintainability index): measures the ease with which we can maintain software or how easily we can make changes to the software; and high is considered as better (measure of maintainability)
* NOC (number of children): measures the number of direct descendants for each class, and classes with large number of children are considered to be more difficult to modify and usually require more testing (measure of efficiency and reusability)
* DIT (depth of inheritance tree): the maximum length from the class node to the root of the tree, and is measured by the number of ancestor classes (measure of reusability and complexity)

**Limitations**: It doesn’t show a benchmark for “good quality” software, many of the results seems generalized, and it doesn’t consider things like bug testing.

**Note:** All the metrics used showed a loss in software quality over the history of versions, and it is especially important that there testing utilized an OSS called JFreeChart.

**Important Note**: An article like this should be a main focus of interest because it’s lacking a lot of things that we could consider. It doesn’t consider the community involvement in OSS or security measurements, and actually limits itself in that it’s only performed on a single piece of software.

**Title**: Using Pirate Metrics to measure success of open source projects

**Link**: <https://opensource.com/business/16/6/pirate-metrics>

**Summary**:

The article discusses the use of “Pirate Metrics” (redeveloped from another model) to measure community involvement and retention in open source software projects. It mentions that metrics for community involvement are especially useful for smaller projects to ascertain if it’s heading down a good trajectory. These metrics measure the following:

* Acquisition: A user visits the website or repository
* Activation: A user downloads the product
* Retention: A user opens up an issue in the community
* Referral: A user stars the repository on GitHub (useful user feedback)
* Revenue: A user regularly opens issues or contributes to code

**Limitation**: It may be difficult to measure some of these specific metrics on our end. This idea is advertised as more of a self-reflection framework whereby a developer may analyze their own work.

**Title:** The size of open-source communities and its impact upon activity, licensing, and hosting

**Link**: <http://redmonk.com/dberkholz/2013/04/22/the-size-of-open-source-communities-and-its-impact-upon-activity-licensing-and-hosting/>

**Summary**:

The post makes an interesting discussion on the fact that once a project reaches a certain number of contributors, it appears to behave differently than a smaller project. The author uses a comparative chart/spreadsheet to show that committers tend to be more active, as a whole, in larger projects. These larger projects also tend to become more self-hosted, and pay attention to things like licensing.

**Limitation**: While this is an excellent way of explaining why a larger community can be better, it doesn’t necessarily “prove” that more committers equals a better product.

**Title**: Open Source Metrics

**Link**: <https://opensource.guide/metrics/>

**Summary**:

The article makes a case for measuring certain metrics when creating open source projects. Utilizing metrics helps creators in responding to user feedback, see project popularity, understand how the project is used, etc. The primary reference the author uses is based upon the metric graphs that appear on GitHub projects. They divide the metrics into the following categories:

* Discovery: is the project being seen by people
* Usage: is the project being downloaded/cloned
* Retention: are people regularly committing to the project

**Limitation**: These metrics are quite useful for checking community involvement, however the examples they use are limited to GitHub (they provide minor alternatives). Similar to previous community articles, this is more directed towards actual creators and how they can better their own projects (possible limitation on research application).

**Note**: The final metrics in the “Retention” section have some points on issue turnover rate (how often issues are resolved), which in itself may be another useful metric.

**Title:** QA Metrics: The Value of Testing Metrics Within Software Development

**Link**: <https://www.getzephyr.com/resources/whitepapers/qa-metrics-value-testing-metrics-within-software-development>

**Summary**:

The article makes the point that as a result of shorter deadlines and lower resources, metrics become all the more important. That being said it also explains that metrics are only useful when taken at the whole (only measure certain aspects). The article further states that while there are no universal metrics for all products, there are some important ones which can be divided as followed:

* Project: for the purpose of analysing whether a project is meeting an objective (general)
  + Requirement Coverage: percentage of number of requirements covered divided by the total number of requirements
  + Defect Open and Close Rate: percentage of number of defects found before delivery divided by total defects (defects found before and after)
* Department: provide generalized day-to-day analysis metrics
  + MTTD: number of issues detected over total execution time
  + MTTR: number of issues fixed over total coding time
  + Defect Removal Efficiency: percentage of number of pre-release defects divided by number of total defects
* Company: analyzes delivery to consumer
  + Customer Reported Issue Percentage: percentage of total number of defects found by customer divided by total number of product defects

**Limitation**: Getting some of the data from a few of these metrics may be difficult, or even impossible. Some the metrics imply knowledge of day-to-day activities which may limit the result.

**Note**: I refrained from mentioning some of the metrics (especially those in the “Company” section) since they either apply to multiple projects together or are impossible to test without restricted data.