**Full Article Summaries**

**Name**: Joseph Tassone

**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. It draws the conclusion that software becomes more complex over a series of releases, and quality therefore decreases over time.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, it doesn’t clearly explain how larger corresponds to lower quality, and doesn’t actually show how it actually tested the metrics. It also performs it’s testing on different versions of the same software which seems strange to me as the end product is what ultimately matters.

**Note:** All the metrics used showed a loss in software quality over the history of versions, and it is especially important that their testing utilized an OSS called JFreeChart. A metric like MI could be especially useful since it’s a combination of a couple metrics.

**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. We could actually mention what we utilize for testing, and improve upon their flawed conclusion.

**Title**: Comparing Software Metric Tools

**Link**: <http://dl.acm.org.libproxy.auc.ca/citation.cfm?id=1390648&CFID=942414749&CFTOKEN=24620810>

**Summary**:

The article expresses the fact that different metric tools interpret and implement the definitions of software differently. Basically, what this means is that the tools could have heavy variation even though they are showing the exact same metrics. The importance of this variation is quite relevant to scientific study where we wish to get as accurate of results as possible. The primary purpose of the article is to answer two questions:

1. Do metric values of given software systems and metric definitions depend on the metric tools used to compute them (confirmed in the results)?
2. Does the interpretation of metrics values of given software system as induced by a quality mode depend on the metrics tool (partially confirmed).

The article goes on to consider some limitations in their study which involves lack of universality among tools, the actual cost/availability of tools, the fact that not all tools measure the same thing, and that there’s no “gold standard” for valid metrics. For the most part they adhere to their self-imposed limitations and their results are quite clear in that there is heavy variation. The article states that much of this comes down to scope and what the tools consider in regards to the extent of the individual metrics. As an example, a few of the tools consider built in API objects and comments for things like LOC and inheritance tree. The following are the metrics that were tested on three different pieces of software:

* CBO (coupling between object classes)
* DIT (depth of inheritance tree)
* LCOM-CK (lack of cohesion of methods)
* LCOM-HS (lack of cohesion of methods)
* LOC (lines of code)
* NOC (number of children)
* NOM (number of methods)
* RFC (response for a class)
* WMC (weighted methods per class)

It should also be noted that ten metric tools were tested out as this presented the largest subset of available tested metrics.

**Limitation**: The article doesn’t really answer whether a metric itself is useful or relevant, and focuses on OOD. It also has the issue of scope which was mentioned earlier in the summary.

**Note**: It points out important notes that should be considered when doing our testing, like the fact that we can’t rely on using multiple tools. It doesn’t say that the fact that they different results is a bad thing, but more that these results can’t be relied on if we’re using multiple tools. It should also be noted that they had much higher variations on larger software.

**Important Note**: The article refers to the ISO-9126 quality model and states that to adhere to the model there should be at least one metric relating to couple, cohesion, size, and inheritance. Tests should clearly be run multiple times as variation in results are potential.

**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. It’s also designed from a business framework, and the question of actual

**Note**: It has some interesting notes regarding community, which seems to be a common metric for addressing open source quality (indirect quality). The consensus seems that the more activations you have, the more the implied open source product can be viewed as having “good quality”.

**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. An interesting point is made that the vast majority of active projects have few actual committers, and actual trend shows a linear decrease when compared to projects over number of committers. Basically, projects with a high number of community members are in the vast minority when compared to projects with few. 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. The conclusion is that a larger project (based on community) behaves much more actively and progressively than one which is small by comparison.

**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. He also utilizes an external dataset from Ohloh which he himself admits has limitations. It makes a case for licensing, which isn’t a proven enough discussion point in correlation to quality.

**Note**: An interesting trend is shown here in that the vast majority of projects have a smaller connection of actual committers. This is an interesting fact as it makes logical sense, especially in that the number of lower quality projects should be in the majority. Another idea to consider is that since large community based projects have more involvement, error resolution should theoretically be improved and therefore the actual quality should be greater (needs validation).

**Important Note**: The article makes a clear case why larger projects are generally more successful, or at the very least perform different. This definitely makes a case for a community based software metric, and also why it’s important to consider. It also shows that GitHub is still an excellent choice for choosing projects, as it has gotten over some of the inherent issues of other sites like Sourceforge (for large projects).

**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.

**Important Note**: Community involvement, as later articles point out, is a clear indication of success. We should consider some type of metric here when performing our tests (downloads correspond to quality). GitHub stars may also be useful for our testing grounds as they don’t measure quality, but can allow us to get an initial assumption for picking a variation of OSS.

**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. The other issue are that these metrics are mainly to be used by developers, and may need altering for our purposes.

**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.

**Important Note**: The more relevant metric for our cause may be the “Defect Open and Close Rate”, as the information should be readily available and is a clear sign of error handling (external quality at maintenance level).

**Title**: Effects of the Number of Developers on Code Quality in Open Source Software

**Link**: <http://dl.acm.org.libproxy.auc.ca/citation.cfm?id=1852864&CFID=942414749&CFTOKEN=24620810>

**Summary**:

The article uses a number of metrics to test whether the quality is effected by the number of developers. They state the motivation of this study is driven by the fact that many people suggest quality is directly affected by more or less developers. Their results seem to show that for their given test subjects, there is no correlation. They use the following metrics for testing:

* Cyclomatic Complexity
* Lines of Code per Function
* Comment Density
* Maximum Nesting

**Limitation**: The article doesn’t actually show how testing was performed, and doesn’t reveal their findings. There’s also a limitation to their premise that the number of developers doesn’t affect quality, as their scope seems too small to justify.

**Note**: This is a useful work for contradictory purposes, and also shows that some of the metrics discussed earlier are commonly used.

**Important Note**: This is one of the first articles to discuss comment density, which it states should be about 30%. This is a metric that should be considered for static quality, and maintainability.

**Title**: Evaluating the Quality of Open Source Software

**Link**: <http://www.sciencedirect.com/science/article/pii/S1571066109000632>

**Summary**:

This research paper makes that point that most prior research into quality has been either kept under wraps by an organization or used techniques that were too narrow in focus. With the rise of OSS this issue has changed, as software can now be assessed transparently. As such, they use this piece as a way of presenting motivating examples, tools, and techniques to evaluate to quality of OSS. Specifically the paper presents a technical overview of their SQO-OSS technique, and relates it with other research ventures that exist in this area. The project is said to have evolved from a study utilizing the Maintainability Index, and moved into full scale design after they realized they could develop a system that both calculates and integrates metrics from various product and process-related sources (different from many other simple tools). The software they use for this is of their own design, and is known as Alitheia, which collects the raw data from OSS and makes computations for the metrics. They then move into a study on how product and process metrics can provide insights into software quality, and specifically show that neither a decrease in quality or increase in errors are reported as a result of OSS. In the same section they also show that projects tend to become less erratic after an introductory development period, and then show some other tool and quality models. They conclude by giving an overview of the quality model the created as a result of this project (discussed in another article). In summary, the model uses a hierarchy tree, and separates everything based on code quality or community quality. The following is a list of the associate metrics for their model:

Analyzability

* Cyclomatic number
* Number of statements
* Comments frequency
* Average size of statements
* Weighted methods per class (wmc)
* Number of base classes
* Class comments frequency

Changeability

* Average size of statements
* Vocabulary frequency
* Number of unconditional jumps
* Number of nested levels
* Coupling between objects (cbo)
* Lack of cohesion (lcom)
* Depth of inheritance tree (dit)

Stability

* Number of unconditional jumps
* Number of entry nodes
* Number of exit nodes
* Directly called components
* Number of children (noc)
* Coupling between objects (cbo)
* Depth of inheritance tree (dit)

Testability

* Number of exits of conditional structs
* Cyclomatic number
* Number of nested levels
* Number of unconditional jumps
* Response for a class (rfc)
* Average cyclomatic complexity per method
* Number of children (noc)
* Maturity Number of open critical bugs in the last 6 months
* Number of open bugs in the last six months

Effectiveness

* Number of critical bugs fixed in the last 6 months
* Number of bugs fixed in the last 6 months
* Security Null dereferences
* Undefined values

Mailing list

* Number of unique subscribers
* Number of messages in user/support list per month
* Number of messages in developers list per month
* Average thread depth

Documentation

* Available documentation documents
* Update frequency

Developer base

* Rate of developer intake
* Rate of developer turnover
* Growth in active developers
* Quality of individual developers

**Limitation**: There tool itself seems to be in a locked development state, and isn’t really implementable for our purposes. They are specifically looking at creating new metrics/tools with their product, while we are looking for existing ones. The article also concentrates primarily on other sources, without delving deep enough into their own design and implementation.

**Note**: The article presents many different metrics from other groups, and many points on the use of quality metrics. At the very least the article can be used as an informative reference into the field of OSS quality.

**Important Note**: While much of the article can be simply used from a simple reference point, the concentration from here should be on the model presented (many useful metrics).

**Title**: A Review of Models for Evaluating Quality in Open Source Software

**Link**: <http://ac.els-cdn.com/S2212667813000178/1-s2.0-S2212667813000178-main.pdf?_tid=3ad6e9a8-4d3f-11e7-88af-00000aab0f01&acdnat=1497032142_c5242803f892da854bde0b6aef97397d>

**Summary**:

Quality is something which should be considered when picking OSS for certain solutions. The article recognizes that metrics are a useful way of analysis, and explains how these are group into larger techniques known as models. The authors explain that OSS models came into existence due to the inability of traditional software models not considering aspects like community. From here they compare/contrast six quality models based on different criteria. The models in question which are analysed are:

* CapGemini Open Source Maturity Model
* QSOS
* OpenBRR
* SQO-OSS
* OpenSource Maturity Model
* QualOSS

Using these quality models, they identify and analyse the unique strength of each. They conclude that many of these models were either improvements on each other or based off of classical software models like the ISO/IEC 9126.

**Limitation**: The article limits itself to six models, and doesn’t go into depth on what the models themselves consider.

**Note**: This is a useful jumping off point into some of the models. It also allows us to consider which ones to look into more in-depth, as we shouldn’t be seeking metrics without some form of implementation (example: tools).

**Important Note:** The models for further analysis are SQO-OSS, QualOSS, and the OpenSource Maturity Model.

**Title**: The SQO-OSS quality model: measurement based open source software evaluation

**Link**: <https://www.dmst.aueb.gr/dds/pubs/conf/2008-OSS-qmodel/html/SGSS08.htm>

**Summary**:

The article discusses how most models are based on a hierarchal structure that measures different aspects related to quality. The authors take steps to present a hierarchical model that evaluates both source code and community driven processes, which they believe are missing from normal modes of analysis. A main point they make is that OSS is becoming far more relevant and therefore proper methodology should be developed to adapt to this phenomenon. The model under discussion is mainly metric oriented, and describes metrics as directly measurable attributes of software. The authors go forward with pointing out how they differ from other models:

* Criticizes other models like OSMM for even though they are simple to implement, they don’t consider things like source code
* Criticizes OpenBRR method for even though it includes community, it considers the reference application a weak idea
* QSOS allows for objective results among users, but lacks flexibility

The SQO-OSS utilizes continuous monitoring, focuses on automation (no heavy user interference), assesses source code, and considers OSS community. The authors believe that while the other models do part of these, they don’t necessarily do all of them (or if they do not well). The model consists of two phases: definition of the evaluation model and definition of the aggregation method (data collection). It can be expanded

* Break down source code analysis into the hierarchy: analyzability, changeability, stability, testability, maturity, effectiveness, security
* Break community down into: mailing list, documentation, developer base

Weighting from poor to excellent is used to assess quality, and then grading is performed up the hierarchy (main categories must match).

**Limitation**: The article and the model make the dangerous assumption that all metrics are of equal importance, and at the same time contradict themselves by pointing out the limitation of certain metrics. The structure is based on automation, however this seems limiting as human intervention should at least be an option (it’s a limited option). Another issue is the fact that it doesn’t consider usability as it requires human intervention. They also point out the limitation of models themselves, which seems counter intuitive to the argument for them. Finally, their community metrics are redundant and could easily be simplified down.

**Note**: This article seems to be overly complex, however it could be used as a contrast as to why models may not be the best choice for quality.

**Important Note**: They make a good case for why we can’t actually analyze functionality reliably because it implies we know the prime directives of the creators.

**Title**: The QualOSS Open Source Assessment Model

**Link**: <https://www.rose-hulman.edu/class/csse/OldFiles/csse575/Resources/MeasOpenSource-05314237.pdf>

**Summary**: Similar to traditional software, the quality assessment of open source software must also be analyzed. A difference from primary models directed towards closed source is that the model takes into account product quality, as well as process maturity and sustainability of the underlying community. The article points out the advantage of OSS in that it doesn’t necessarily need to be concerned with licensing costs and allows for greater independence between software vendors. However, even considering theses advantages, picking one is risky without assessing and could create a negative impact for an organization. Assessment of OSS is a comprehensive process and involves both robustness and evolvabiltiy. The authors critique the fact that while many other models generally cover relevant data, they have a very rudimentary perspective and a lack of coverage. The primary focus of QualOSS is composed of quality characteristics, metrics, and indicators. It should be noted that metrics correspond to concrete aspects which are themselves, measurable. Quality characteristics are based on a hierarchy model and are organized into two levels of characteristics and subcharacteristics:

* + Product related: maintainability, reliability, transferability, operability, performance, functional suitability, security, compatibility
  + Community related: maintenance capacity, sustainability, process maturity

What truly separate this concept from closed source are the development communities behind the OSS. Finally, the assessment process itself includes change submissions and review, peer review of changes, propose significant enhancements, report and handle issues with the product, test the program produced by the project, plan releases, release new versions of the product backport corrections in the current release to previous stable releases.

**Limitation**: The point of evolvability is itself limited as certain pieces of information may not be readily available for normal use. The article itself is limited as it doesn’t actually get specific with metrics or processes. It seems as if (at least at the time of the articles writing) that it hasn’t been fully tested yet, and is still in a highly theoretical stage.

**Note**: Points out an important advantage of open source in that it doesn’t need to be concerned with things like licensing. It also points continues the common these in the inclusion of community involvement.

**Important Note**: Makes a valid point that source code is always available and can be readily analyzed, therefore it should be. It also makes an important point that successful OSS project aren’t rushed or disorganized, and are based on strong communities.

**Title**: Introducing the OpenSource Maturity Model

**Link**: <http://dl.acm.org.libproxy.auc.ca/citation.cfm?id=1572200&CFID=951820277&CFTOKEN=35034244>

**Summary**:

**Limitation**:

**Note**: