1. **Introduction**
2. **Topics**
   1. **Open Source Software**

Compared too many other design phenomenon; Open Source Software (OSS) can be seen as relatively recent or new trend. That’s not to say collaboration between designers hasn’t always been there, and it’s common knowledge that an idea like “free sharing” has existed far longer than most modern technologies. That being said, the actual label of “open source” was first applied at a strategy session in Palo Alto, shortly after the announcement for the release of the Netscape source code [0.1]. The session was inspired by the idea that this release had created an opportunity to discuss and advocate a design process that as of that point wasn’t being openly discussed. As a result of this announcement, many groups began to adopt this new methodology and a community began to build.

In general OSS is software where the source code is made openly available to the public [0.2]. As such anyone can inspect, modify, or even further enhance the existing code. This isn’t necessarily limited to simple inspection, and can involve greater development and cooperation among designers. Basically OSS can considered any software whereby a user can download, change, and then redistribute the software back to the prime repository. It doesn’t generally concern itself with concepts like licensing as it is believed this allows for greater independence between software vendors [6]. The concept isn’t necessarily universal, but is more often the commonplace than the exception. This in itself makes it very different from “classical” software, which not only restricts its source code, but also holds tight licensing among its products. The idea mentioned here can be further extended to the point that OSS clearly has a very different development process than traditional software (example: Waterfall Design Model).

The process itself is relatively simple, but becomes more sophisticated as the project increases in size [11]. Generally, an individual or group will have an idea, and will begin basic development on it. They will then put the source code online, and allow individuals to edit and make changes to the already existing code. In some cases an entire community is created upon this, and the design process begins to refine itself. Error reports or new feature requests will be posted, and contributors will make alterations to fill these requests. The rise of version control systems (example: GitHub) have greatly simplified this process, and have allowed OSS to flourish. One can now find a large variety of projects among OSS, which can include:

* Operating Systems: UNIX/POSIX
* Web servers: Apache and JAWS
* Compilers: GNU C/C++
* Editors: GNU emacs

**2.2 Quality of Software**

Quality can generally be seen as the standard of something determined by the measure against something similar. In the case of software, it’s the measure of a particular software design compared to other types or minimum accepted values. These measurements and comparisons vary greatly, and as of yet there is currently no universal standard [1]. There are also debates on when software quality analysis should take place, as it’s beneficial for different users at different times. The entire process should be considered for designers, while for end-users the result is really the only concern. For the latter, the average user only considers whether the product performs the tasks they require. This corresponds to maintainability, which is associated with the final product and sometimes the only consideration for quality (synonymous with each other).

In terms of quality assessment, there are two commonly used practices for analysis: models and metrics. Models tend to be based on the hierarchal design structure, and usually take inspiration from the ISO9126 design standards [5]. In this case models tend to be more useful for designers as they analyze quality from a multitude of perspectives throughout the entire creation process. If based of the ISO9126 then they tend to consider:

* Functionality
* Reliability
* Usability
* Maintainability
* Portability
* Efficiency

Clearly many aspects play off of each other, and a precise picture is painted. Metrics on the other hand are specific measurements for analysis, and can be used by anybody [12]. It’s important to note that metrics are usually part of the greater process for models, but can also be used as separate calculations. To be clear on the separation; models don’t always lead to direct measurements, whereas metrics are themselves direct measurements.

Now as implied earlier quality is not a fixed and universal property of software. It depends greatly on the context and goals of its stakeholders, and generally one needs to be clear and precise with the quality specification. Overtime, quality of software changes and if the software does not adjust accordingly, its quality will decay [0.3, I...II]. The only real solution to such a problem is continuous quality control, and the use of models or metrics are generally a way to combat this problem. The advantage of OSS becomes clear here, as it utilizes continuous releases and designers from many different backgrounds to resolve errors.

**2.3 Metrics for Measuring Open Source Software Quality**

Software metrics are measurements which act as a scale for representing certain aspects of the software [1]. As mentioned previously, they are direct measurements and are useful for determining whether a specific design corresponds to particular characteristics. Metrics aren’t limited to closed software either, and can be applied to OSS. The reason is that static code analysis doesn’t change for OSS source code, and the resulting software isn’t ultimately any different [2]. In laymen’s terms, the product is the basically the same as any other piece of software from a physical perspective.

Utilizing metrics has many advantages, and can help designers in better responding to feedback [4]. In the case of fast operating environments, with shorter deadlines and less resources (like most OSS projects) metrics become all the more important. The reason for such a case is that metrics can analyze many aspects of a project, and help designers determine where they are faulting [3]. Metrics are mainly used from a static code point, but also can analyze errors and even documentation. While this is incredibly useful, metrics also have a setback in that most don’t have a “gold standard” for acceptable values [12]. There are benchmarks, however most haven’t been widely accepted and it’s recommended to test only those which are widely documented.

Another issue is that many research papers don’t seem to agree on when analysis should occur (example: the design process or final product). Models make the case for the doing it throughout the process, and this will be touched on in the next section. A point is made in a paper on the SQO-OSS quality model on the difficult of properly analyzing functionality, as one cannot know the prime directives of the creators [7]. This further limits certain metrics, as they become only useful to designers and not end-users themselves. One clear consensus exists and that is towards the analysis of source code. For OSS, the justification is that source code it is always available, therefore it should be analyzed [6]. The ISO9216 quality standard is often used as a complete reference for the looking at the quality of software. However, it’s also been stressed that much of is redundant and repetitive. To be safe it is suggested in one article to at the very least utilize at least one metric for coupling, cohesion, size, and inheritance to be safe [12].

OSS itself allows for software quality to be examined transparently and without many barriers [2]. The problem here is that while static code analysis remains the same, there are issues analyzing the big picture. Basically, there aren’t commonly accepted metrics that consider enough for it to be fully useful for OSS. A major example is community which will be discussed in its own section later on. Some metrics which are greatly documented and commonly used are:

* Lines of Code: represents the approximate number of lines of code, which when greater indicates a higher difficulty of maintainability.
* Cyclomatic Complexity: measures how structurally complex the code is, and is a measure of the different number of path flows in the program.
* Maintainability Index: considers other metrics like cyclomatic complexity and lines of code, to determine how maintainable a piece of software is (between 0 and 100).
* Depth of Inheritance: indicates the number of classes that extend from the common root, and the deeper the more difficult it will be to understand.
* Coupling between Objects: measures the coupling of classes, and is indication of code reuse/maintainability (lower is better).
* Lack of Cohesion of Methods: indicates whether a class represents a single abstraction or multiple abstractions, and is a good indication of whether code should be refactored (high is better).
* Comment Density: a ratio of the number of comments relative to the amount of source code, and is general indication of code readability and maintainability [1][2][10][12].

A complete listing of metrics can be found at an ISO9126 site [0.4].

**2.4 Models for Measuring Open Source Software Quality**

As suggested so far, quality is something which should considered when picking OSS for certain solutions. Metrics have already been discussed as useful way of analysis, but these can be further grouped larger techniques known as models. These concepts aren’t new, however with the rise of OSS previous models proved unreliable for proper study. As a result, OSS models came into existence due to the inability of traditional software models not considering aspects like community [5]. Some of the more popular models include:

* SQO-OSS
* OpenSource Maturity Model
* QualOSS

It’s important to note that the ones mentioned most use some form of hierarchal model, broken down into layers. Each layer works off of each other, but generally has its own unique criteria for assessment which may include some form of measuring (example: metrics). This is partially based on the classical ISO9126, however it’s expanded to included ideas unique to OSS [5]. The remainder of this section will be a discussion on a few of the OSS models, and the issue with models in general.

The SQO-OSS model is a hierarchical model that aims itself at the evaluation of source code and community processes, which the creators believe is missing from normal modes of analysis. The model under discussion is mainly metric oriented, and describes metrics as directly measurable attributes of software [7]. Some of the following points explain how it differs from other models for OSS:

* OSMM: simple to implement, but doesn’t consider source code
* OpenBRR: includes community, but it’s use of reference application is a weak idea
* QSOS: allows for objective results among users, but lacks flexibility

The SQO-OSS has continuous monitoring, focuses on automation (with no heavy user interference), assesses source code, and considers OSS community. Researchers suggest that while most other models utilize at least part of these, they don’t necessarily do all of them. The model consists of two phases: definition of the evaluation model and definition of the aggregation method (data collection). It can be expanded as:

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

Weighting from poor to excellent is used to actually assess quality, and then grading is performed up the chain (the main categories must match) [7].

The QualOSS model is an assessment of whether the quality of OSS is sufficient for the intended purpose, and whether the chances of being maintained and supported in the future, as well as of keeping certain quality standards over time, are sufficiently high [6]. This model claims that it differs from those directed towards closed source in that the model takes into account product quality, as well as process maturity and sustainability of the underlying community. It also points out that since OSS doesn’t necessarily need to be concerned with licensing costs and allows for greater independence between software vendors, analysis on anything related to this should be a concern [6]. The model considers the assessment of OSS as a comprehensive process and that involves both robustness and evolvabiltiy. Articles 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 separates 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 [6].

The OMM model is based on knowledge of the Free/Libre Open Source Software process, and is in fact an evolution on the process itself [8]. It utilizes a tree level scale in order to keep it simple and its goal is to improve the quality of OSS as a whole. The model plays off the idea that the assessment must be done carefully as there is not only a high diversity of OSS projects, but also a high diversity of how they are developed. A problem exists in that other models generally focus primarily on the final product, and not the development process. Relating to this idea there is also the belief that there aren’t enough metrics for the whole development process of OSS. The work done to define the OMM model is composed of two main activities:

* Research of quality expectations on the FLOSS process by all stakeholders (questionnaires and interviews)
* Definition of the OMM based on the Goal Question Metric (GQM) approach (based on the compiled results from the first part)

The model comprises of trustworthy elements grouped into 3 maturity levels named basic, intermediate, and advanced. At the same time it’s organized into levels, with each level building on the previous lower level [8]. Articles argue that any user should be able to utilize this method (industry developers, communities, average users).

A clear advantage of models can be seen in this argument, in that they expand off metrics and consider the larger process of OSS. Community is an idea considered by nearly every model, and is something that should be analyzed when considering quality. That being said, they are clearly not usable by just anybody and if one doesn’t understand the created intention, then much of this is unusable. At the same time, if the individual assessing doesn’t have certain piece of information (limited to developers) then the models again become unusable. Due to these restrictions, models should only be used by those who are directly involved in project, but not necessarily by end-users or other secondary stakeholders.

**2.5 Community Involvement**

As mentioned previously, a major difference between OSS and closed source are the potential development communities. Since the source code is available for download and updating, developers from across the world can become involved. As such, there are cases where a large group consolidate and work cooperatively on a single project. That being said, research suggests that the vast majority of active projects have few actual committers, and actual trends show a linear decrease when compared to projects over number of committers. In fact, once a project reaches a certain size it begins to act quite differently than smaller ones, and takes a more progressive approach [11].

The research conducted by different groups seems divided on whether this increase in project size directly corresponds to an increase in quality. One paper performed tests on a set of projects utilizing a variety of metrics, but found no decrease in quality based solely on community size [10]. This idea makes sense as it’s completely subjective to those working on the project. A better measurement for community might be the idea of how active a community is. There’s a lot of supportive research for this, as having members isn’t necessarily enough and “retention” is much useful [4][9]. This idea considers not just whether there is a large community, but how often issues are being resolved and how much participation there is. This may not directly show whether something has high quality, but does show how often errors are being resolved and could explain bad quality.

The rise of version control repositories has made not only OSS easier, but also has made measuring community involvement more readily available. GitHub is one of the most recognizable of these, and contains many metrics for measuring things like weekly commits. Utilizing these metrics designers can see where they need to improve, and users can determine how active the community for the project is. GitHub also allows users to “star” projects they enjoy, which in itself doesn’t explain whether a project is high quality, but does justify whether a project has a positive feedback.

**2.6 Quality Testing Methods and Tools**

A point on testing must be made, as there are many tools for the use of quality assessment. However, a research paper expressed the fact that different metric tools interpret and implement the definitions of software differently [12]. Basically, what this means is that all the tools used 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 same research goes on to consider some limitations in the 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 [12]. The last point is especially worth considering as it not only means multiple metric tools can be used for testing, it also means that testing has to be comparative among multiple projects. Basically, for there to be proper analysis of quality (at this point in time), multiple projects must be tested for a baseline to be developed.

1. **Conclusion and Research Proposal**
   1. **Conclusion**

As discussed, software evaluation isn’t a new trend, however with the rise of open source software current methods aren’t necessarily sufficient. Quality is something that should, and must be evaluated in order to pick the correct solution to a problem. Procedures like metric utilization and models are an excellent choices to solving this particular issue, however they each have their own unique problems. Models are limiting in that they’re only useful for designers with access to restricted qualitative data, and most end-users are only concerned with the final product. Therefore concept which should be the point of concentration is the idea of “maintainability”. If a product is easy to maintain, it should be easier to use and at the same time have a better quality. There are many metrics directed to this concept which paint the big picture for those querying the project. However, metrics for community must also be considered as they may not necessarily explain good quality, they most certainly can explain bad quality. The justification for that is through showing how active a project is, and how often things are resolved. Finally, testing for metrics must also be carefully done as many tools are not comparable, and standardized baselines have yet to be set.

* 1. **Research Proposal**

# There are a lot of researches on the quality of traditionally developed software or Closed Source Software (CSS). OSS are said to be possibly of comparable quality with CSS. However, there are very few studies investigating common software quality factors for various types of Open Source projects [12]. Open Source Projects have their own characteristics which might affect their quality:

a) Different levels of programmers. Open Source Project is joined by different levels of people who are interested in the open source project. In Closed Source Project, programmers are well organized in one organization.

b) Frequent beta releases. A cornerstone of open-source is short feedback loops between users and core developers, which typically result in frequent “beta” releases, e.g., several times a month. Although this schedule satisfies the end-users who receive quick patches for bugs they found in earlier betas, it can be frustrating to other end-users who want more stable, less frequent software releases.

c) Platforms-independence. Another cornerstone of open-source software is its platform-independence, which stems from its roots in the open systems—rather than proprietary systems—community. Support for platform-independence, however, can yield the task of keeping an open-source source software base operational despite continuous changes to the underlying operating system and compiler platforms.

d) Support for many compile-time and run-time configurations. The availability of the software in open-source projects encourages core developers to increase the number of options for configuring and subsetting the software at compile-time and run-time. Although this flexibility enhances the software’s applicability for a broad range of use-cases, it can also greatly magnify QA costs due to the combinatorial number of code paths that must be regression tested.

As the OSS paradigm makes progress within these organizations any potential software procurer is tasked with some important questions which, currently, cannot be answered with any real assurance:

* Many OSS projects are very similar. How do we choose between them? Which is the most appropriate system for the company’s IT infrastructure?
* How can we distinguish the “good” and “bad” projects?
* How can we reason about the quality of a software product in order to trust its future development?

Unfortunately these organizations often have nothing more than word-of-mouth on which to base their judgments of OSS products. With 1097071 projects currently hosted on SourceForge it is understandable that products of excellent quality may be overlooked. It is possible to supplement the word-of-mouth tradition with more formal metrics to evaluate the OSS.

**Objective(s) or Activities**

### The objectives of this project include:

# Investigate suitable metrics to measure important software quality factors and choose those suitable for OSS

# Select a range of representative Open Source projects for measurement

# Evaluate the software of these projects according to the selected quality factors

# Using statistic tools to analyse and present the result

* Models limit themselves to designers, while metrics allow for use by anyone
  + At the same time they also produce useful results that give a “big picture” and non redundant answer
* To keep on point we will add community based metrics, as well as those than analyze documentation to ensure maintainability
* Mention GitHub stars

**References**