Curating GitHub for Engineered Software Projects

Nuthan Munaiah, Steven Kroh, Craig Cabrey, Meiyappan Nagappan
{nm6061, skk8768, cac2573}@rit.edu, mei@se.rit.edu
Software Engineering Department
Rochester Institute of Technology
Rochester, NY 14623

ABSTRACT

Software forges like GitHub host millions of repositories. Software engineering researchers have been able to take advantage of such a large corpora of potential study subjects with the help of tools like GHTorrent and Boa. However, the simplicity in querying comes with a caveat: there are limited means of separating the signal (e.g. repositories containing software engineering projects) from the noise (e.g. repositories containing home work assignments). The proportion of noise in a random sample of repositories could skew the study and may lead to researchers reaching unrealistic, potentially inaccurate, conclusions. We argue that it is imperative to have the ability to sieve out the noise in such large repository stores.

We propose a framework, and present a reference implementation of the framework as a tool called `reaper`, to enable researchers to select GitHub repositories that contain evidence of an engineered software project. We identify software engineering practices and the propose means of validating their existence in a GitHub repository. `reaper` analyzed 2,247,382 GitHub repositories and identified 250,362 repositories as containing engineered software projects. We compared the performance of `reaper` to other de-noising criteria, such as the number of GitHub stargazers, in evaluating a random sample of 768 repositories. We found the stargazer-based criteria to exhibit high precision (100%) but an inversely proportional recall (0.69%). On the other hand, we found `reaper` to exhibit a high precision (76.56%) and a high recall (67.90%). The stargazer-based criteria offers precision but it fails to represent a significant portion of the population.

1. INTRODUCTION

Software repositories contain a wealth of information about the code, people, and processes that go into the development of a software project. Retrospective analysis of these software repositories can yield valuable insights into the evolution and growth of the software projects contained within. We can trace such analysis all the way back to the 1970s, when Belady et al. proposed Lehman's Laws of software evolution [20]. Today, the field is significantly invested in retrospective analysis with the Boa project receiving more than $1.4 million to support such analysis[1].

The insights gained through retrospective analysis can effect the decision-making process in a project, and improve the quality of the software system being developed. An example of this can be seen in the recommendations made by Bird et al. in their study of the effects of code ownership on the quality of software systems [21]. The authors suggest that quality assurance efforts should focus on those components with many minor contributors.

The richness of the data and the potential insights that it represents were the enabling factors in the inception of an entire field of research–Mining Software Repositories (MSR). In the early days of MSR, researchers had limited access to software repositories, that were primarily hosted within organizations. However, with the proliferation of open access software repositories such as GitHub, Bitbucket, SourceForge, and CodePlex for source code; and Bugzilla, Mantis, and Trac for bugs, researchers now have an abundance of data from which to mine and draw interesting conclusions.

Every source code commit contains a wealth of information that can be used to gain an understanding of the art of software development. For example, Eick et al. dived into the rich (fifteen-plus year) commit history of a large telephone switching system in order to explore the idea of code decay [29]. Furthermore, modern day source code repositories provide features that make managing a software project as seamless as possible. While the integration of features provides improved traceability for developers and project managers, it also provides MSR researchers with a single, self-contained, organized, and more importantly, publicly-accessible source of information from which to mine. However, anyone may create a repository for any purpose at no cost. Therefore, the quality of information contained within the forges may be diminishing with the addition of many noisy repositories e.g. repositories containing home work assignments, text files, images, or worse, a repository used as a backup store for a desktop. Kalliamvakou et al. identified this noise as one of the nine perils to be aware of when mining GitHub data for software engineering research [33]. The situation is compounded by the sheer volume of repositories contained in these forges. GitHub alone hosts over 31 million repositories [9] and this number is rapidly increasing.

Researchers have used various criteria to slice the mam-

FSE '16 November 13–19, 2016, Seattle, WA, USA

DOI: 10.1145/1235

[1]NSF Grant CNS-1513263

moth software forges into data sets manageable for their studies. For example, MSR researchers have leveraged simple filters such as popularity to remove such repository noise. Filters like popularity (measured as number of watchers or stargazers on GitHub, for example) are merely proxies and may neither be general-purpose nor representative of an engineered software project. Furthermore, MSR researchers should not have to reinvent filters to eliminate unwanted repositories. There are a few examples of research that take the approach of developing their own filters in order to procure a data set to analyze:

- In a study of the relationship between programming languages and code quality, Ray et al. selected 50 most popular (measured by the number of *stars*) repositories in each of the 19 most popular languages [39].
- Bissyandé, Tegawendé F., et al. chose the first 100,000 repositories returned by the GitHub API in their study of the popularity, interoperability, and impact of programming languages [23].
- Allamanis et al. chose the 14,807 Java repositories with at least one fork in their study of applying language modeling to mining source code repositories [19].
- The project sites for GHTorrent [6] and Boa [13] list more papers that employ different filtering schemes.

The assumption that one could make is that the repositories sampled in these studies contain engineered software projects. However, source code forges are rife with repositories that do not contain source code, let alone an engineered software project. Kalliamvakou et al. manually sampled 434 repositories from GitHub and found that only 63.4% (275) of them were for software development; the remaining 159 repositories were for experimental, storage, academic, empty, or no longer accessible [33]. The inclusion of repositories containing such non-software artifacts in studies targeting software projects could lead to conclusions that may not be applicable to software engineering at large. At the same time, selecting a sample by manual investigation is not a feasible approach given the sheer volume of repositories hosted by these source code forges.

The goal of our work is to identify practices that an engineered software project would typically exhibit with the intention of developing a generalizable framework with which to identify such projects in the real-world. The primary contributions of our work are:

- A generalizable evaluation framework defined on a set of dimensions that encapsulate typical software engineering practices;
- A reference implementation of the evaluation framework, called `reaper`, made available as an open-source project [16];
- A publicly-accessible list of GitHub repositories containing engineered software projects, as determined by the `reaper` [5].

2. ENGINEERED SOFTWARE PROJECT

We define an engineered software project as follows:

Definition 1. *Engineered software project—a software project that leverages sound engineering practices in each of its dimensions, e.g., documentation, testing, and project management.*

The dimensions of an engineered software project may correspond with software development life cycle activities of which there are many. We have chosen a list of eight dimensions for use with our evaluation framework.

2.1 Evaluation Framework

In order to operationalize Definition 1, we need to (a) identify the essential software engineering practices that are employed in the building and maintenance of a typical software project, and (b) propose means of quantifying the evidence of their use in a software project. The *evaluation framework* is our attempt at achieving this goal.

The evaluation framework takes the form of a function as depicted in Equation 1. Essentially, the evaluation framework produces a score quantifying the extent to which the content of a repository represents an engineered software project.

$$\text{score}(r) = \sum_{d \in D} (M_d \geq t_d) \times w_d \tag{1}$$

Where,

- r is the repository to evaluate.
- D is a set of dimensions along which the repository, r, is evaluated. These are analogous to the software engineering practices e.g., unit testing, documenting source code, etc.
- M_d is the metric that quantifies evidence of the repository, r, employing a certain software engineering practice in the dimension, d. For example, the proportion of comment lines to source lines quantifies documentation.
- t_d is a threshold that must be satisfied by the corresponding metric, M_d, for the repository, r, to be considered engineered along the dimension, d. For example, having a sufficiently high proportion of comment lines to source lines indicates the project is engineered along the dimension, say documentation.
- $M_d \geq t_d$ evaluates to 1 if the metric value, M_d, satisfies the corresponding threshold requirement, t_d, or 0 otherwise.
- w_d is the weight that specifies the relative importance of each dimension d.

A key consideration in the design of the evaluation framework was flexibility. Equation 1 is flexible enough to be tailored to operationalize an alternative definition of an engineered software project by using a different set of dimensions, D, and corresponding metrics, thresholds, and weights.

In this study, we have defined an engineered software project along the following eight dimensions:

1. *Architecture*, as evidence of code organization.
2. *Community*, as evidence of collaboration.
3. *Continuous Integration*, as evidence of quality.
4. *Documentation*, as evidence of maintainability.
5. *History*, as evidence of sustained evolution.
6. *Issues*, as evidence of project management.
7. *License*, as evidence of accountability.
8. *Unit Testing*, as evidence of quality.

These dimensions were chosen after considering the trade-off between simplicity and accuracy. Algorithms to measure each dimension must be generic enough to account for the plethora of programming languages that source code repositories contain and yet, be specific enough to produce meaningful results. We acknowledge that this list is subjective, and by no means exhaustive; however, Equation 1 is flexible enough to accommodate the addition of any other dimension.

3. IMPLEMENTATION

In this section, we describe the reference implementation of our framework, called `reaper`. `reaper` is an open-source project and its source code is available on GitHub at https://github.com/RepoReapers/reaper. In its current version, the capabilities of `reaper` are limited owing to the following restrictions:

- The repository being evaluated must be publicly accessible on GitHub.
- The primary language of the repository must be one of Java, Python, PHP, Ruby, C++, C, or C#. We choose these languages based on their popularity on GitHub as reported by GitHut [10].

3.1 Data Sources

In this section, we elaborate on the two sources of data used in our study. Each dimension of a repository that we chose to evaluate may require the use of either or both data sources.

3.1.1 Data Source: GitHub and Git Metadata

GitHub metadata contains a wealth of information with which we could describe several phenomena surrounding a source code repository. For example, some of the important pieces of metadata are the primary language of implementation in a repository and the commits made by developers to a repository.

GitHub provides a REST API [8] with which GitHub metadata may be obtained over the Internet. There are several services that capture and publish this metadata in bulk, avoiding the latency of the official API. The GitHub Archive project was created for this purpose. It stores public events from the GitHub timeline and publishes them via Google BigQuery. Google BigQuery is a hosted querying engine that supports SQL-like constructs for querying large data sets. However, accessing the GitHub Archive data set via BigQuery incurs a cost per TB of data processed.

Fortunately, Gousios et al. have a free solution in their GHTorrent Project [30]. The GHTorrent project provides a scalable, queriable, and offline mirror of all Git and GitHub metadata available through the GitHub REST API. The GHTorrent project is similar to the GitHub Archive project in that both start with the GitHub's public events timeline. While the GitHub Archive project simply records the details of a GitHub event, the GHTorrent project exhaustively retrieves the contents of the event and stores the them in a relational database. Furthermore, the GHTorrent data sets are available for download, either as incremental MongoDB dumps or a single MySQL dump, allowing offline access to the metadata. We have chosen to use the MySQL dump which was downloaded and restored on to a local server. In the remainder of the paper, we use the term database to refers to the GHTorrent database.

The database dump used in this study was released on April 1, 2015. The database dump contained metadata for 16,331,225 GitHub repositories. As mentioned earlier, our implementation is restricted to repositories in which the primary language is one of Java, Python, PHP, Ruby, C++, C, or C#. Furthermore, we do not consider repositories that have been marked as deleted and those that are forks of other repositories. Deleted repositories restrict the amount of data available for the analysis while forked repositories can artificially inflate the results by introducing near duplicates into the sample. With these restrictions applied, the size of our sample is reduced to 2,247,382 repositories.

An inherent limitation of the database is staleness of data. There may be repositories in the database that no longer exist on GitHub as it may have been: 1. deleted, 2. renamed, 3. made private, or 4. blocked by GitHub.

3.1.2 Data Source: Repository Contents

Since `reaper` can only evaluate publicly accessible repositories hosted by GitHub, these repositories are our primary data source. Developers typically interact with their repositories using either the `git` client or the GitHub web interface. Developers may also use the GitHub REST API to programmatically interact with GitHub.

`reaper` uses GitHub to obtain a copy of the source code for each repository that it evaluates. We cannot use GitHub's REST API to retrieve repository snapshots, as the API uses `git archive` to create those snapshots. In this case, the snapshot may not include files the developer has marked irrelevant to an end user (such as unit test files). We want to examine all development files in our analysis, so we use `git clone` instead to ensure all files are downloaded.

As mentioned earlier, the metadata used in this study is current as of April 1, 2015. However, this metadata may not match a repository cloned after 1 April 2015, as the repository owner may have made commits after that date. In order to synchronize the repository with the metadata, we reset the state to a past date. For each evaluated repository, we retrieved the `date` of the most recent commit to the repository. We then identified the `SHA` of the last commit made to the repository before the end of the day identified by `date` using the command `git log -1 -before="{date} 11:59:59"`.

For repositories with no commits recorded in the database, we used the date when the GHTorrent metadata dump was released i.e. 2015-04-01. We identified the appropriate commit `SHA`, and ensured the state of the cloned repository was reset using the command `git reset --hard {SHA}`.

3.2 Dimensions

In this section, we elaborate on each of the eight dimensions that were introduced in the Section 2.1 earlier. In each subsection, we describe the attribute of an engineered software project that the dimension represents, propose a metric to quantify the dimension, and describe an approach to collect the metric from a GitHub repository.

3.2.1 Architecture

IEEE 1471 defines software architecture as "the fundamental organization of a system embodied in its components, their relationships to each other and to the environment,

and the principles guiding its design and evolution" [34]. In effect, software architecture is the high-level structure of a software system that depicts the relationships between various components that compose the system.

Software architecture comes in all shapes, sizes, and complexities. There is no one fixed architecture to which all software systems adhere; however, software systems that employ some form of architecture have discernible relationships between components that compose the system. These components can be as granular as functions and as coarse as entire binaries. For our purposes, any software system that has well-defined relationships between its components can be said to have an architecture. The presence of an architecture indicates that some form of design was employed in the development of the software project. Consequently, the software project would contain further evidence of being an engineered software project.

Metric We approximate repository architecture as an undirected graph in which nodes represent files in the repository and edges represent relationships between the files. We hypothesize that a well-architected software system, at the very least, will have a majority of its files related to one another. We propose a metric, *monolithicity*, to quantify the extent to which source files in the repository are related to one another.

DEFINITION 2. *Monolithicity—the ratio of the number of nodes in the largest connected subgraph to the number of nodes in the entire architecture graph.*

There is a specific case that must be handled when calculating *monolithicity*. For example, certain projects may have only one source file present in the repository. Zazworka et al. investigated the impact of using god classes in the architecture of a software system and found that their inclusion has a detrimental effect on that system's quality [45]. Thus, we consider a repository with a single source file to have no architecture.

Approach Bowman et al. designed a methodology to extract the high level architecture of the Linux kernel [24]. The researchers began by grouping source files into modules based on naming and directory structure. Individual source file relationships were promoted to relationships between the modules to produce an architectural overview of the Linux kernel. We use a similar approach to reverse engineer software architecture from the source code of a software system.

We use a popular syntax highlighter package called Pygments [14] to identify symbol definition and usage in source files. Although Pygments' primary use case is as a syntax highlighter, the lexer that powers it internally is suited to our purposes. Furthermore, Pygments supports lexical analysis of over 300 programming languages ensuring our implementation can scale beyond the languages chosen in this study.

A repository may contain files with source code in multiple programming languages. We use `ack` [1] to obtain a list of files that contain code in the primary language of the repository (from the database). We then follow a two pass approach to construct the architecture graph.

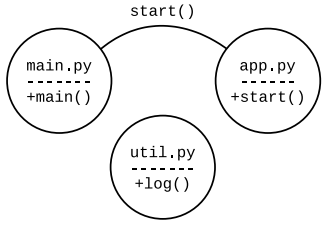
1. Pygments lexically analyzes each source file identified by `ack` to collect the symbols defined and referenced by the file. The file, along with a list of its symbol definitions and references, is added as a node to the graph.
2. Iterating through all the nodes in the graph, an edge is added between a node A, and a node B, if at least one symbol referenced by A is defined by B.

Figure 1 shows the architecture graph constructed from an example repository containing three source files: `main.py`, `app.py`, and `util.py`. Each node displays a file name above a list of externally visible symbols. The edges represent cross-file symbol references. The monolithicity of this graph is ($\frac{2}{3} = 0.66$), as the largest connected component has two files (`main.py` and `app.py`).

While simple, this approach does have limitations, notably:

- Dynamic Loading: Lack of support for programming languages that support dynamic loading e.g. JavaScript. JavaScript is an extremely dynamic language, which makes it difficult to determine symbol references among source files purely from analyzing the contents of the source files.
- Over-approximation: Since the approach relies purely on the name of the symbols, the resulting architecture may depict relationships that may not occur in the source code e.g., when multiple files define symbols with the same name.
- Language Bias: Since the approach only considers files containing source code in the primary language of the repository, there is a possibility whereby a significant portion of source code in the repository may be excluded. E.g., A repository meant for the development of a Django application may be flagged with JavaScript being the primary language if the percentage of JavaScript code is higher than that of Python, perhaps by the inclusion of vendor files such as jquery.js or bootstrap.js in the repository.

Figure 1: Example view of an extracted architecture



One of the implementation concerns of our approach is the computational complexity to generate the architecture graph is $O(n^3)$.In the case of some very large projects, the approach was not a feasible method to get a result in a reasonable amount of time. Therefore, the reference implementation includes a timeout period. If the tool is unable to calculate the *monolithicity* of a project within the designated time period, the calculation is halted and the repository receives a zero score for the architecture dimension.

3.2.2 Community

Software engineering is an inherently collaborative discipline. With the advent of the Internet and the plethora of tools that simplify communication, software development is increasingly decentralized. Open source development in particular thrives on decentralization, with globally dispersed

developers contributing code, knowledge, and ideas to a multitude of open source projects. Collaboration in open source software development manifests itself as a community of developers. The presence of a developer community indicates that there is some form of collaboration and cooperation involved in the development of the software system, which is partial evidence for the repository containing an engineered software project.

Metric Whitehead et al. have hypothesized that the development of a software system involving more than one developer can be considered as an instance of collaborative software engineering [43]. We propose a metric, *core contributors*, to quantify the community established around a source code repository.

DEFINITION 3. *Core contributors—the cardinality of the smallest set of contributors whose total number of commits to a source code repository accounts for 80% or more of the total contributions.*

Approach The notion of *core contributors* is prevalent in open source software where a set of contributors take ownership of and drive a project towards a common goal. Mockus et al. have applied this concept in their study of open source software development practices [35]. The definition of core contributors is the same as that of core developers [41] as defined by Syer et al.

We computed total contributions by counting the number of commits made to a repository as recorded in the database. We then grouped the commits by author and picked the first n authors for which the cumulative number of commits accounted for 80% of the total contributions. The value of n represents the *core contributors* metric.

There is one issue in the implementation of this metric in `reaper`. We use the GHTorrent data to find unique contributors to a repository. However, GHTorrent has the notion of "fake users" who do not have GitHub accounts [7] but publish their contributions with the help of real GitHub users. For example, a "fake user" makes a commit to a local Git repository, then a "real user" pushes those commits to GitHub using their account. Sometimes, the real and fake users may be the same. This is the case when a developer with a GitHub account makes commits with a secondary email address. This tends to inflate the *core contributors* metric for small repositories with only one real contributor, and could be improved in the future by detecting similar email addresses.

3.2.3 Continuous Integration

Continuous integration (CI) is a software engineering practice in which developers regularly build, run, and test their code combined with code from other developers. CI is done to ensure that the stability of the system as a whole is not impacted by changes. It typically involves compiling the software system, executing automated unit tests, analyzing system quality and deploying the software system.

With millions of developers contributing to thousands of source code repositories, the practice of continuously integrating changes ensures that the software system contained within these constantly evolving source code repositories is stable for development and/or release. The use of CI is further evidence that the software project might be considered an engineered software project.

Metric If the project uses a CI service, $M_{\text{CI}} = 1$, otherwise $M_{\text{CI}} = 0$.

Approach The use of a continuous integration service is determined by looking for a configuration file (required by certain CI services) in the source code repository. An inherent limitation of this approach is that it supports the identification of stateless CI services only. Integration with stateful services such as Jenkins, Atlassian Bamboo, and Cloudship cannot be identified since there may no trace of the integration in the repository. The continuous integration services currently supported are: Travis CI, Hound, Appveyor, Shippable, MagnumCI, Solano, CircleCI, and Wercker.

3.2.4 Documentation

Software developers create and maintain various forms of documentation. Some forms are part of the source files, such as code comments, whereas others are external to source files, such as wikis, requirements, and design documents. One purpose of documentation is to aid the comprehension of the software system for maintenance purposes. Among the many forms of documentation, source code comments were found to be most important, second only to the source code itself [27]. The presence of documentation, in a sufficient quantity, indicates the author thought of maintainability; this serves as partial evidence towards a determination that the software system is engineered.

Metric In this study, we restrict ourselves to documentation in the form of source code comments. We propose a metric, *comment ratio*, to quantify a repository's extent of source code documentation.

DEFINITION 4. *Comment ratio—the ratio of the number of comment lines of code (cloc) to the number of non-blank lines of source code (sloc) in a repository r.*

$$M_D(r) = \frac{cloc}{sloc + cloc} \tag{2}$$

Approach We use a popular Perl utility `cloc` [3] to compute source lines of code and comment lines of code. `cloc` returns blank, comment, and source lines of code grouped by the different programming languages in the repository. We aggregate the values returned by `cloc` when computing the comment ratio.

We note that comment ratio only quantifies the extent of source code documentation exhibited by a repository. We do not consider the quality, staleness, or relevancy of the documentation.

3.2.5 History

Eick et al. have shown that source code must undergo continual change to thwart feature starvation and remain marketable [29]. A change could be a bug fix, feature addition, preventive maintenance, vulnerability resolution, etc. The presence of sustained change indicates that the software system is being modified to ensure its viability. This is partial evidence towards a determination that the software system is engineered.

Metric In the context of a source code repository, a commit is the unit by which change can be quantified. We propose a metric, *commit frequency*, to be the frequency by which a repository is undergoing change.

DEFINITION 5. *Commit frequency—the average number of commits per month.*

$$M_H(r) = \frac{1}{m} \sum_{i=1}^{m} c_i \tag{3}$$

Where,

- c_i is the number of commits for the month i
- m is the number of months between the first and last commit to the repository r

Approach Each c_i was computed by counting the number of commits recorded in the database for the month i. However, m was computed as the difference in months between the date of the first commit and date of the last commit to the repository. If m was computed to be 0, the value of the metric was set to 0.

3.2.6 Issues

Over the years, there have been a plethora of tools developed to simplify the management of large software projects. These tools support some of the most important activities in software engineering such as management of requirements, schedules, tasks, defects, and releases. We hypothesize that a software project that employs project management tools is representative of an engineered software project. Thus, the evidence of the use of project management tools in a source code repository may indicate that the software system contained within is engineered.

There are several commercial enterprise tools available, however, there is no unified way in which these tools integrate with a source code repository. Source code repositories hosted on GitHub can leverage a deceptively named feature of GitHub—GitHub Issues—to potentially manage the entire lifecycle of a software project. We say deceptively named because an "issue" on GitHub may be associated with a variety of customizable labels which could alter the interpretation of the issue. For example, developers could create user stories as GitHub issues and label them as *User Story*. The richness and flexibility of the GitHub Issues feature has fueled the development of several third party services such as Codetree[4], HuBoard [11], waffle.io [17], and ZenHub [18]. These services use GitHub Issues to support lifecycle management of projects.

Metric In this study, we assume the sustained use of the GitHub Issues feature to be indicative of management in a source code repository. We propose a metric, *issue frequency*, to quantify the sustained use of GitHub Issues in a repository.

Definition 6. *Issue frequency—the average number of issue events transpired per month.*

$$M_I(r) = \frac{1}{m}\sum_{i=1}^{m} s_i \tag{4}$$

Where,

- s_i is the number of issues events for the month i
- m is the number of months between the first and last commit to the repository r

Approach Each s_i was computed by counting the number of issue events recorded in the database for the month i. However, m was computed as the difference in months between the date of the first commit and date of the last commit to the repository. If m was computed to be 0, the value of the metric was set to 0.

An inherent limitation in the approach is that it does not support the discovery of other project management tools. Integration with other project management tools may not be easy to detect because structured links to these tools may not exist in the repository source code.

3.2.7 License

A user's right to use, modify, and/or redistribute a piece of software is dictated by the license that accompanies the software. Licenses are especially important in the context of open source projects as an article [25] by The Software Freedom Law Center discusses. The article highlights the need for and best practices in licensing open source software.

A software with no accompanying license is typically protected by default copyright laws, which state that the author retains all rights to the source code [12]. Although there is no legal requirement to include a license in a source code repository, it is considered a best practice. Furthermore, the terms of service agreement of source forges such as GitHub may allow publicly viewable repositories to be forked (copied) by other users. Thus, including a license in the repository explicitly dictates the rights, or lack thereof, of the user making copies of the repository. The presence of a software license is necessary but not sufficient to indicate a repository contains an engineered software project according to our definition of the dimension.

Metric If the project has a license, $M_{\mathrm{L}} = 1$, otherwise, $M_{\mathrm{L}} = 0$.

Approach The presence of a license in a source code repository is assessed using the GitHub License API. The license API identifies the presence of popular open source licenses by analyzing files such as `LICENSE` and `COPYING` in the root of the source code repository.

The GitHub License API is limited in its capabilities in that it does not consider license information contained in `README.md` or in source code files. Furthermore, the API is still in "developer preview" and as a result may be unreliable. On the other hand, any improvements in the capabilities of the API is automatically reflected in our approach. In the interim, however, we have overcome some of the limitations by analyzing the files in a source code repository for license information. We identify license information by searching repository files for excerpts from the license text of 12 popular open source licenses. For example, we search for "The MIT License (MIT)" to detect the presence of The MIT License. The 12 chosen licenses were enumerated by the GitHub License API (https://api.github.com/licenses). We note that the interim solution implemented may have its own side-effect in cases where a repository, with no license of its own, includes source code files of an external library instead of defining the library as a dependency. If any of the external library source code files contain excerpts of the license we search for, the license dimension may falsely indicate the repository to contain a license.

Since license is a critical aspect of a source code repository, we consider it to be an *essential* dimension, i.e. the absence of a License in a repository automatically evaluates the score to 0.

3.2.8 Unit Testing

An engineered product is assumed to function as designed for the duration of its lifetime. This assumption is supported by the subjection of the product to rigorous testing. An engineered *software* product is no different in that the guarantee of the product functioning as designed is provided by rigorous testing. Evidence of testing in a software project implies that the developers have spent the time and effort to ensure that the product adheres to its intended behavior. However, the mere presence of testing is not a sufficient mea-

sure to conclude that the software project is engineered. The adequacy of tests is to be taken into consideration as well. Adequacy of the tests contained within a software project may be measured in several ways [46]. Metrics that quantify test adequacy by measuring the *coverage* achieved when the tests are executed are commonly used. Essentially, collecting coverage metrics requires the execution of the unit tests which may in-turn require satisfying all the dependencies that the program under test may have. Fortunately, there are means of approximating adequacy of tests in a software project through static analysis. Nagappan, et al. have used the number of test cases per source line of code and number of assertions per source line of code in assessing the test quantity in Java projects [37]. Additionally, Zaidman et al. have shown that test coverage is positively correlated with the percentage of test code in the system [44].

Metric We propose a metric, *test ratio*, to quantify the extent of unit testing effort.

DEFINITION 7. *Test ratio—the ratio of number of source lines of code in test files to the number of source lines of code in all source files.*

$$M_U(r) = \frac{slotc}{sloc} \tag{5}$$

Where,

- *slotc* is the number of source lines of code in test files in the repository r
- *sloc* is the number of source lines of code in all source files in the repository r

Approach In order to compute *slotc*, we must first identify the test files. We achieved this by searching for language- and testing framework-specific patterns in the repository. For example, test files in a Python project that uses the native unit testing framework may be identified by searching for these patterns:

- `import unittest`
- `from unittest import TestCase`

We used `grep` to search for and obtain a list of files that contain specific patterns such as the above. We then use the `cloc` tool to compute *sloc* from all source files in the repository and *slotc* from the test files identified. Occasionally, a software project may use multiple unit testing frameworks e.g. a Django web application project may use Python's `unittest` framework and Django's extension of `unittest–django.test`. In order to account for this scenario, we accumulate the test files identified using patterns for multiple language-specific unit testing frameworks before computing *slotc*.

The multitude of unit testing frameworks available for each of the programming languages considered makes the approach limited in its capabilities. We currently support 20 unit testing frameworks. The unit testing frameworks currently supported are: Boost, Catch, googletest, and Stout gtest for C++; clar, GLib Testing, and picotest for C; NUnit, Visual Studio Testing, and xUnit for C#; JUnit and TestNG for Java; PHPUnit for PHP; django.test, nose, and unittest for Python; and minitest, RSpec, and Ruby Unit Testing for Ruby.

In the event we are unable to identify a unit testing framework, we resort to considering all files in directories named test, tests, or spec as test files.

3.3 Thresholds

In the previous section, we described the eight dimensions along which a source code repository is evaluated. In this section, we describe the process of establishing a threshold value for the metric associated with each dimension.

Recall from Section 2.1 that a threshold associated with a metric is a numerical value that must be satisfied by the metric. Given a repository and a dimension, if the dimension's metric satisfies its threshold, we consider the repository engineered along that dimension.

The thresholds used in our implementation were established using a set of 150 GitHub repositories that were known to contain engineered software projects. We selected these 150 repositories manually by looking at repositories owned by organizations such as Amazon, Apache, Facebook, Google, and Microsoft. We refer to this set as the "representative set" in the remainder of this paper.

We computed our eight metrics for each repository in the representative set. Outliers were eliminated using the Peirce criterion [40]. For the Boolean–valued metrics, 1 is the threshold. For all other metrics, the corresponding threshold was chosen to be the minimum of the metric values in the representative set.

The value of the thresholds for each metric is presented in Table 1. We used R version 3.2.1 to compute the thresholds and the Peirce package to perform outlier elimination [38] [26].

3.4 Weights

In this section, we present a weighting scheme that was used in our study. Recall from Section 2.1 that a weight is a numeric value that defines the relative importance of a dimension in the evaluation framework.

The weights chosen in our study are presented in Table 1. We note these weights are subjective. When assigning each weight, we also considered the limitations in collecting the value of the associated metric from a source code repository. For example, the approach to evaluating a source code repository along the architecture dimension is more robust and thus its weight is higher than the unit testing dimension, where there are inherent limitations owing to our non-exhaustive set of framework signatures.

An oddity in the weights from Table 1 is that the license dimension, though being the most critical, has a weight of zero. The reason for this is that we consider license to be an essential dimension, i.e. the absence of a License in a repository automatically evaluates the entire score to 0.

3.5 Reference Score

At this point, we have defined all the elements necessary to evaluate the score of a repository as defined by Equation 1. In order to assess if a repository contains an engineered software project we needed to establish a base against which to compare the score of a candidate repository. We call this base the *reference score*.

DEFINITION 8. *Reference score—the minimum score that a repository must achieve in order to be considered to contain an engineered software project.*

Any repository scoring fewer points than the weakest repository from the reference set in Section 3.3 should not be considered an engineered software project. Therefore, we established the reference score as the minimum score achieved by a repository in the reference set. We calculated the reference score as 60.

Table 1: Dimension Weights and Metric Thresholds

Dimension (d)	Weight (w_d)	Metric (M_d)	Threshold (t_d)	Interpretation
Architecture	20	Monolithicity	0.649123	At least 64.9123% of source files must be connected to one another in the largest subsystem.
Community	20	Core Contributors	2	At least 2 contributors whose commits account for 80% of the total commits.
Continuous Integration	5	Evidence of CI	1	Evidence of continuous integration usage must be present.
Documentation	20	Comment Ratio	0.001866	At least 0.1866% of the source lines must be comments.
History	20	Commit Frequency	2.089552	At least 2.089552 commits per month.
Issues	5	Issue Frequency	0.022989	At least 0.022989 issues per month.
License	0	Evidence of License	1	Evidence of a license usage must be present.
Unit Test	10	Test Ratio	0.001016	At least 0.1016% of source lines must be unit test code.

4. RESULTS

We ran `reaper` on a sample of 2,247,382 GitHub repositories. The tool determined 250,362 ($\approx$ 11%) of the repositories to contain engineered software projects per Definition 1. We have analyzed individual repository scores, and found a significant majority of scores are zero. The bulk of these repositories received a zero score due to the absence of a license. Thus, we recomputed the scores for all repositories omitting the license requirement. In this case, the number of engineered software projects increased to 439,535 ($\approx$ 20%).

The results of the evaluation may be viewed at https://reporeapers.github.io. The results include the metric values collected from each repository as well as the score assigned to the repository. Again, we consider a repository to contain an engineered software project when its score is greater than or equal to 60 i.e. the reference score. The website allows the entire data set to be downloaded as a CSV file. We envision the data set will help researchers select GitHub repositories for their MSR data sets.

5. VALIDATION

Here, we describe the methodology followed to validate `reaper`. We considered validation from two perspectives: *internal*, in which the classification performance of `reaper` was validated, and *external*, in which the classification performance of `reaper` was compared to that of a classification scheme that uses number of stargazers [39] as the criteria. We used false positive rate (FPR), false negative rate (FNR), precision, recall, and F-measure to assess the classification performance.

5.1 Establishing the Ground Truth

In order to evaluate the performance of a classification scheme, the ground truth classification of the samples in a validation set must be known. Assuming a 5% confidence interval (margin of error) and 95% confidence level (degree of certainty), we randomly sampled 384 repositories that had a `reaper` score greater than or equal to 60 and 384 repositories that had a `reaper` score less than 60 [15]. Our validation set, therefore, was composed of 768 repositories. We manually investigated the 768 repositories to classify each repository as containing an engineered software project, or otherwise. We used the label "project" to indicate that a repository contained an engineered software project and the label "not-a-project" to indicate otherwise. To earn the "project" label, we required a repository to contain software that could have utility for others besides the author, regardless of popularity. The only artifacts considered in labeling a repository were the repository description, README file, commit log, documentation, Wiki, or source code comments.

In order to ensure an unbiased evaluation, the validation set was independently evaluated by two authors (Author #1 and Author #2). We acknowledge that the manual evaluation may have been biased by the authors' prior knowledge of `reaper`. However, as mentioned earlier, we restricted ourselves to assessing the utility of the project contained in a repository rather than manually replicating the functionality implemented by `reaper`. As an example, `reaper` could not detect a license in the repository `brandur/casseo`; therefore the tool did not label the repository as a project. However, the project is a ruby gem that has utility for others. It has been downloaded more than 11,000 times [2]. In our manual evaluation, we labeled `brandur/casseo` as a project based on its demonstrated utility.

The repositories in the validation set were ordered such that there was no discernible pattern and the ordering was different for the two authors. With the manual evaluation from both authors tabulated, disagreements, if any, were settled. There were 178 (out of the 768) repositories that were classified differently by the two authors. These repositories were discussed and a consensus was reached with Author #1 having to alter 114 labels and Author #2 having to alter 64 labels. We consider the outcome of the manual evaluation to be the "ground truth" classification of the repositories in the validation set.

The list of repositories in the validation set along with the labels (by Author #1, Author #2, and effective), `reaper` score, number of stargazers, outcome of classification based on `reaper` score, and outcome of classification based on num-

ber of stargazers is available for download as a CSV file at https://reporeapers.github.io/static/validation.csv.

5.2 Internal

The performance of the classification scheme using `reaper` score is shown in Table **??**. The tool has a 26.87% false positive rate when run with the thresholds and weights described in Table 1. This means, about a fourth of the time, it incorrectly labels repositories as projects. With the same settings, the tool has a 32.10% false negative rate. Again, about a third of the time, `reaper` incorrectly labels repositories as non-projects.

Table 2: Performance of `reaper`-based Classification Scheme Against the Ground Truth

FPR	FNR	Precision	Recall	F-measure
26.87%	32.10%	76.56%	67.90%	71.97%

We manually inspected one instance of a repository being falsely labeled as a project and one of being falsely labeled as not-a-project to reason about the probable causes for the mis-classification . The details of the inspection are outlined below:

False Positive: The repository `sigmarkarl/world` merely contains a collection of seemingly unrelated Java projects, but it was falsely labeled as a project by `reaper`. An inspection of the metric values computed by `reaper` revealed that the Java projects have sufficient documentation, commit history, and architecture to score higher than the reference score.[2] It just so happens that the repository isn't meant to be consumed by anyone else—it is a collection or backup of personal code. This highlights one of `reaper`'s shortcoming: a well-written personal code, that is not necessarily meant for outside use, may (objectively) be misclassified as a project.

False Negative: The repository `mgkimsal/zfkit` was falsely labeled as not-a-project by `reaper`. An inspection of the metric values computed by `reaper` revealed that `reaper` underestimated the community and history dimensions as a consequence of a potential bug in the GHTorrent data causing the number of commits to be lesser than that shown on GitHub. This highlights another of `reaper`'s shortcomings: the system is much more complex than the naïve, stargazers-based approach, in that it attempts to dissect many dimensions of repositories written in many languages. `reaper` may under report a dimension based upon a bug in the `reaper`'s implementation or a bug in one of its dependencies. In this vein, visit `reaper`'s GitHub site for a list of known bugs: https://github.com/reporeapers/reaper/issues.

5.3 External

MSR research reveals how software engineering is applied across many projects. Useful MSR data sets therefore include repositories reflective of software engineering work and ignore repositories that may be noise. The goal of our approach is to identify repositories that have a high likelihood of containing an engineered software project while ignoring repositories that could be noise. To validate our approach, we must compare the performance of `reaper` and current practices in generating useful MSR data sets with the ground truth.

We previously noted repository popularity is one potential criterion in identifying a data set for MSR studies. The intuition is popular repositories (with many "stargazers") will contain actual software that people like and use [32]. This intuition is the basis for several well-received studies. For example Ray, et al.'s work on on programming languages and code quality (which has gained 34 citations) [39] and Guzman, et al.'s paper on commit comment sentiment analysis (which has been cited 15 times) [31] have used the number of stargazers as a way to select projects for their case studies[3]. These papers use the top starred projects in various languages, which are bound to be extremely popular. For example, Ray, et al.'s dataset includes `mongodb/mongo`, which has 8,927 stars [39].

In Table 2 from the previous Section, we presented the performance metrics of evaluating `reaper` against the ground truth. We now use the stargazers-based classification scheme to evaluate the repositories from the validation set. We used a generous threshold of 500 stars as an indicator of popularity. We say 500 is generous because out of the 1,940,692 repositories for which we have stargazers data for, 3,968 repositories with at least 500 stars are approximately in the 99.79^{th} percentile. The performance metrics from the stargazers-based scheme are shown in Table 3.

Table 3: Performance of stargazers-based Classification Scheme Against the Ground Truth

FPR	FNR	Precision	Recall	F-measure
0.00%	99.31%	100.00%	0.69 %	1.38%

As seen in Table 3, the stargazers-based approach has a 0% false positive rate but an almost 100% false negative rate. While a repository with a large number of stars is likely to contain an engineered software project, the contrary is not always true. Therefore, by using the stargazers-based approach, researchers may be excluding a large set of repositories that contain engineered software projects but may not be popular.

We must also compare our approach and the stargazers-based approach to demonstrate external validity. In Table 4, we segment the repositories by their ground truth classifications. For those repositories we manually classified as projects, we list the percentage of repositories where (a) both approaches (`reaper` and the number of stargazers) agree, (b) both approaches disagree, (c) `reaper`'s result matches the ground truth but stargazers does not, and (d) the stargazers approach matches the ground truth but `reaper`'s does not. We also present the performance metrics for the segment of repositories classified as not-a-project in our ground truth.

For repositories we believe are useful in MSR data sets, both approaches incorrectly classify the repositories as non-projects 32.10% of the time—neither approach is perfect. However, `reaper` correctly classifies 67.21% of the repositories correctly when the stars approach does not. This means `reaper` is more lenient in classifying projects, deciding to include less popular but still relevant projects.

[2]The community metric is over reported as discussed in Section 3.2.2. Omitting community, the repository is still classified a project.

[3]Citation counts retrieved from Google Scholar

Table 4: Comparison of `reaper` and Stars Classification Schemes for Repositories with Different Ground Truth Labels

Label: project

Agree	Disagree	`reaper` Agrees	Stars Agree
0.69%	32.10%	67.21%	0.00%

Label: not-a-project

Agree	Disagree	`reaper` Agrees	Stars Agree
73.13%	0.00%	0.00%	26.87%

A perfect example is `jruby/jruby-ldap`, from the team that maintains the `ruby` implementation on the Java Virtual Machine (JVM). The repository contains a ruby gem for LDAP support in jruby. Our tool classifies the repository as a project due to its architecture, commit history, test suite, and documentation, among other factors. However, the repository has only 7 stars. While the stargazers approach misses `jruby/jruby-ldap`, this project could be useful in an MSR dataset. We also note that there is no case where the stargazers approach classifies something as a project, which `reaper` does not. Hence, any repository classified as a project by the stargazers approach is also classified the same by `reaper`.

In the other segment, 73.13% of the time, both approaches correctly classify non-projects. In addition, the stars approach correctly classifies repositories as non-projects 26.87% of the time when `reaper` fails to do so.

Consider the repository `Stabledog/bin-pub`, which has sufficient documentation, commit history, and architecture to satisfy `reaper`'s threshold, however, the repository is essentially a collection of configuration files for a UNIX user account, not incredibly useful in a general mining study. The stargazers approach classifies `Stabledog/bin-pub` as not-a-project only for its lack of stars.

Summary: From the data, we make three conclusions about the suitability of `reaper` to help MSR researchers generate useful data sets. First, the strict stargazers approach ignores many valid projects, but enjoys a 0% false positive rate. This means popular repositories on GitHub most likely contain real software. Second, `reaper` is able to correctly classify many "unpopular" projects. This means the `reaper` framework can extend the population from which sample data sets are drawn, hopefully improving the external validity of the conclusions from MSR research. Third, we conclude that `reaper` is imperfect, introducing false positive projects into MSR data sets. Perhaps, `reaper` could be used as an initial selection criteria augmented by the stargazers approach. Nevertheless, we have shown that more work can be done to improve data collection methods in MSR research projects that will in-turn improve the accuracy of `reaper`.

6. RELATED WORK

An early work by Nagappan [36] reveals opportunities and challenges in studying open source repositories in an empirical context. Kalliamvakou et al. describe various perils of mining GitHub data; specifically, Peril IV is: "A large portion of repositories are not for software development" [33]. Dyer, et al. and Bissyandé et al. have created domain specific languages Boa [28] and Orion [22], respectively, to help researchers mine data about software repositories. Jarczyk et al. defines two measures of quality for GitHub software projects [32], one dealing with popularity of the project while the other is based on how fast the project team resolves issues.

Ohloh.net (now Black Duck Open Hub) is a publicly editable directory of free and open source software projects. The directory is curated by Open Hub users, much like a public wiki, resulting in an accurate and up-to-date directory of open source software. Researchers [42] have used Open Hub to search for repositories containing engineered software projects as perceived by Open Hub users.

7. CONCLUSION

The goal of our work was to understand the elements that constitute an engineered software project. In the pursuit of this goal, we proposed an operational definition of an engineered software project across eight dimensions and built an evaluation framework to reduce the GitHub population to the set of engineered software projects. Each dimension represents an attribute of a GitHub repository and has an associated metric to quantify the attribute. The dimensions are: architecture, community, continuous integration, documentation, history, issues, license, and unit testing. We provide, `reaper`, a reference implementation of the framework available as an open-source project on GitHub [16].

We manually curated a representative set of 150 GitHub repositories that we knew to contain engineered software projects. We applied our evaluation framework to the representative set and computed threshold values for the metrics associated with each of the eight dimensions. We also established a reference score from the representative set. We then evaluated a sample of 2,247,382 GitHub repositories to identify the repositories containing engineered software projects. The evaluation revealed that a very small percentage ($\approx$ 11%) of scored GitHub repositories in our sample actually contain engineered software projects. This result implies Peril IV in Kalliamvakou, et al. does not go far enough [33]. They found 37% of repositories in their sample were not for software development (using a definition less strict than *engineered software project*). This means MSR researchers must be particularly careful when selecting criteria to automatically filter large samples of GitHub repositories.

We acknowledge that the outcome of the evaluation may be biased towards our definition of an engineered software project. The evaluation framework and the reference implementation were designed to be flexible enough to support alternate definitions of an engineered software project. We encourage the research community to contribute to `reaper` and to use this work as a starting point for choosing MSR datasets.

Acknowledgments

We thank Nimish Parikh for his contributions during the initial stages of the research. We thank our peers for providing us with their GitHub authentication keys which helped us attain the API bandwidth required. We also thank the Research Computing Group at Rochester Institute of Technology for providing us the computing resources necessary to execute a project of this scale.

8. REFERENCES

[1] Beyond grep: ack 2.14, a source code search tool for programmers. http://beyondgrep.com/. Accessed: 2016-03-11, Version: 2.14.

[2] casseo | RubyGems.org | your community gem host. https://rubygems.org/gems/casseo/. Accessed: 2016-03-11.

[3] CLOC – Count Lines of Code. http://cloc.sourceforge.net/. Accessed: 2016-03-11, Version: 1.62.

[4] Codetree - GitHub Issues, Managed. https://codetree.com/. Accessed: 2016-03-11.

[5] Engineered software projects (evaluated by `reaper`). https://reporeapers.github.io/static/downloads/projects.csv. Accessed: 2016-03-1.

[6] GHTorrent Hall of Fame. http://ghtorrent.org/halloffame.html. Accessed: 2016-03-11.

[7] GHTorrent: The relational DB schema. http://ghtorrent.org/relational.html. Accessed: 2016-03-11.

[8] GitHub API v3 | GitHub Developer Guide. https://developer.github.com/v3/. Accessed: 2016-03-11.

[9] GitHub Press. https://github.com/about/press. Accessed: 2016-03-11.

[10] GitHut - Programming Languages and GitHub. http://githut.info. Accessed: 2016-03-11.

[11] HuBoard - GitHub issues made awesome. https://huboard.com/. Accessed: 2016-03-11.

[12] No License - Choose a License. http://choosealicense.com/no-license/. Accessed: 2016-03-11.

[13] Publications Related to Boa - Boa - Iowa State University. http://boa.cs.iastate.edu/papers/. Accessed: 2016-03-11.

[14] Pygments: Python Syntax Highlighter. http://pygments.org/. Accessed: 2016-03-11.

[15] Sample Size Calculator - Confidence Level, Confidence Interval, Sample Size, Population Size, Relevant Population - Creative Research Systems. http://www.surveysystem.com/sscalc.htm. Accessed: 2016-03-11.

[16] `reaper` - reference implementation. https://github.com/RepoReapers/reaper. Accessed: 2016-03-11.

[17] Waffle.io - Work Better on GitHub Issues. https://waffle.io/. Accessed: 2016-03-11.

[18] ZenHub - Project Management for Agile Teams on GitHub. https://www.zenhub.io/. Accessed: 2016-03-11.

[19] M. Allamanis and C. Sutton. Mining source code repositories at massive scale using language modeling. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 207–216. IEEE, 2013.

[20] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems journal*, 15(3):225–252, 1976.

[21] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[22] T. Bissyande, F. Thung, D. Lo, L. Jiang, and L. Reveillere. Orion: A software project search engine with integrated diverse software artifacts. In *Engineering of Complex Computer Systems (ICECCS), 2013 18th International Conference on*, pages 242–245, July 2013.

[23] T. F. Bissyandé, F. Thung, D. Lo, L. Jiang, and L. Réveillere. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*, pages 303–312. IEEE, 2013.

[24] I. T. Bowman, R. C. Holt, and N. V. Brewster. Linux as a case study: Its extracted software architecture. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 555–563, New York, NY, USA, 1999. ACM.

[25] S. F. L. Center. Managing copyright information within a free software project, 2012.

[26] C. Dardis. *Peirce: Functions for removing outliers, with illustrations*, 2012. R package version 0.5/r2.

[27] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd Annual International Conference on Design of Communication: Documenting & Designing for Pervasive Information*, SIGDOC '05, pages 68–75, New York, NY, USA, 2005. ACM.

[28] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *35th International Conference on Software Engineering*, ICSE'13, pages 422–431, May 2013.

[29] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *Software Engineering, IEEE Transactions on*, 27(1):1–12, 2001.

[30] G. Gousios. The ghtorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, Piscataway, NJ, USA, 2013. IEEE Press.

[31] E. Guzman, D. Azócar, and Y. Li. Sentiment analysis of commit comments in github: an empirical study. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 352–355. ACM, 2014.

[32] O. Jarczyk, B. Gruszka, S. Jaroszewicz, L. Bukowski, and A. Wierzbicki. Github projects. quality analysis of open-source software. In *Social Informatics*, pages 80–94. Springer, 2014.

[33] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA, 2014. ACM.

[34] M. Maier, D. Emery, and R. Hilliard. Software architecture: introducing ieee standard 1471. *Computer*, 34(4):107–109, Apr 2001.

[35] A. Mockus, R. T. Fielding, and J. Herbsleb. A case study of open source software development: the Apache server. In *Proceedings of the 22nd international conference on Software engineering*, pages 263–272. Acm, 2000.

[36] N. Nagappan. Potential of open source systems as project repositories for empirical studies working group results. In V. Basili, D. Rombach, K. Schneider, B. Kitchenham, D. Pfahl, and R. Selby, editors, *Empirical Software Engineering Issues. Critical Assessment and Future Directions*, volume 4336 of *Lecture Notes in Computer Science*, pages 103–107. Springer Berlin Heidelberg, 2007.

[37] N. Nagappan, L. Williams, J. Osborne, M. Vouk, and P. Abrahamsson. Providing test quality feedback using static source code and automatic test suite metrics. In *Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on*, pages 10–pp. IEEE, 2005.

[38] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2015.

[39] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 155–165. ACM, 2014.

[40] S. M. Ross. Peirce's criterion for the elimination of suspect experimental data. *Journal of Engineering Technology*, 20(2):38–41, 2003.

[41] M. D. Syer, M. Nagappan, A. E. Hassan, and B. Adams. Revisiting prior empirical findings for mobile apps: an empirical case study on the 15 most popular open-source Android apps. In *CASCON*, pages 283–297, 2013.

[42] Y.-H. Tung, C.-J. Chuang, and H.-L. Shan. A framework of code reuse in open source software. In *Network Operations and Management Symposium (APNOMS), 2014 16th Asia-Pacific*, pages 1–6. IEEE, 2014.

[43] J. Whitehead, I. Mistrík, J. Grundy, and A. van der Hoek. Collaborative software engineering: concepts and techniques. In *Collaborative Software Engineering*, pages 1–30. Springer, 2010.

[44] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. Van Deursen. Mining software repositories to study co-evolution of production & test code. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 220–229. IEEE, 2008.

[45] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, pages 17–23, New York, NY, USA, 2011. ACM.

[46] H. Zhu, P. A. Hall, and J. H. May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.