Jari Haapasaari

Impact of Software Reuse on the Software Quality within Go Ecosystem

Master's Thesis in Information Technology

October 20, 2023

University of Jyväskylä

Faculty of Information Technology

Author: Jari Haapasaari

Contact information: haapjari@gmail.com

Supervisor: Prof. Tommi Mikkonen tommi.j.mikkonen@jyu.fi

Title: Impact of Software Reuse on the Software Quality within Go Ecosystem

Työn nimi: Ohjelmistojen uudelleenkäytön vaikutus Go:lla kehitettyjen ohjelmisto-projektien laatuun

Project: Master's Thesis

Study line: Software and Telecommunication Technology

Page count: 46+0

Abstract: Modern software components almost always rely on partly reused code, and modern programming language ecosystems offer tools that make software reuse effortless. Reused software has become such everyday tools that there is a generation of software developers who have solely worked with a software framework, and have little to no understanding whats under the hood.

The world of third-party libraries and software frameworks has led to a design pattern called "Opportunistic Design." Typical traits of opportunistic design are that developers copy and paste different code snippets from online and utilize different libraries and frameworks in a carefree manner, prioritizing convenience over a systematic and abstraction-driven approach. It is also typical that the developer does not fully understand the reused code, as a result of which unknown and unwanted side effects may arise in the developed software.

The thesis examines open-source projects developed in Go, employing statistical research methods, such as hypothesis testing, in order to study whether there are implications of opportunistic design patterns within the ecosystem. The study is conducted by measuring quality in the form of composite variables collected from the metadata of version-controlled projects, calculating correlations between these variables, and used third-party code.

Findings imply that opportunistic design patterns are not visible within the Go ecosystem. This is, however, not a generalizable result because programming language ecosystems are only comparable to a degree. Refine existing composite variables to better represent quality, check whether the relationship with quality is causative, and seek for more variables with causative relationships with software project quality was left as a future research topic.

Keywords: Opportunistic Design, Software Reuse, Software Quality, Go

Suomenkielinen tiivistelmä: Moderni ohjelmistokehitys nojautuu vahvasti ohjelmistojen uudelleenkäyttöön. Lähes jokaisen ohjelmointikielen ekosysteemistä löytyy paketinhallintaohjelmisto, joka helpottaa erilaisten kirjastojen sovittamista kehitettävään projektiin. Ohjelmointikehykset ovat niin arkipäiväistyneitä, että kokemattomammat ohjelmistokehittäjät eivät välttämättä tiedä mitä kehys tai kirjasto abstrahoi käyttäjälle.

Kirjastojen ja kehyksien ekosysteemissä on syntynyt toimintatapa, tai eräänlainen "suunnittelumalli", jota tässä kontekstissa kutsutaan "opportunistiseksi suunnittelumalliksi" (eng. *Opportunistic Design*). Opportunistille suunnittelumallille on tyypillistä, että ohjelmoijat leikkaavat ja liimaavat ja uudelleenkäyttävät huolettomasti erilaisia kirjastoja, priorisoiden käytännöllisyyttä suunnitelmallisen kehittämisen sijaan. Tyypillistä on myös, että kehittäjä ei välttämättä täysin ymmärrä uudelleenkäytettävää koodia tai kirjastoa. Tämä avaa mahdollisuuksia tuntemattomille sivuvaikutuksille kehitettävässä projektissa, ja mahdollisesti vaarantaa projektin tietoturvan.

Tutkielmassa tutustutaan Go -ohjelmointikielellä kehitettyihin avoimen lähdekoodin projekteihin, hyödyntäen tilastollisia tutkimusmenetelmiä, kuten hypoteesitestausta ja pyritään selvittämään onko ekosysteemissä havaittavissa opportunistista käytänteitä. Tutkielmassa pyritään laaduttamaan projektit ja etsiä korrelaatioita suoraan lainatun koodin kokonaismäärään.

Tulokset viittaavat siihen, että näitä käytänteitä ei ole selvästi havaittavista tutkitusta ekosysteemistä. Tämä ei kuitenkaan ole koko totuus, koska ohjelmointikielien ekosysteemit ovat keskenään erilaisia ja korkeintaan temaattisesti vertailukelpoisia. Jatkotutkimukseksi on esimerkiksi ehdotettu kausatiivisten suhteiden etsimistä ohjelmistoprojektien laadun kanssa.

Avainsanat: opportunistinen suunnittelu, ohjelmistojen uudelleenkäyttö, ohjelmistoprojektin laatu, Go

Glossary

API	API or Application Programming Interface is an interface for software applications to integrate or communicate with each others. (RedHat 2023)
CRUD	CRUD (Create, Read, Update, Delete) is an acronym that describes the basic database operations. (Martin 1983)
Dependency	Dependency refers to a relationship between software component, where other component needs other component to provide the functionality. (SonaType 2023)
EDA	Exploratory Data Analysis (EDA) is a technique, or a process, which describes a sequence of steps to obtain insights from the data. EDA has an emphasis to visualize the data. (Madhugiri 2023)
Go	Go is an open-source programming language developed by Google. (Go 2023)
GraphQL	GraphQL is a custom query language for APIs, in order to get only the data client needs. (GraphQL 2023)
Library	Library refers to a collection of software functionality, which is packaged to be reused. (Rouse 2016)
LOC	LOC or SLOC is shorthand for lines of code or source lines of code. Definition refers to a line of code, which is not a comment or blank line. (GeeksForGeeks 2023)
Module	Module refers to a sub-set of logical functionality. In the end software includes multiple software modules. (LawInsider 2023)
Open-Source	Open-Source (OSS) is software, where source code is available online for anyone to inspect, modify or improve. (OpenSource 2023)
ORM	ORM or "Object Relational Mapping" is a procedure or technique to map objects to relational data structures, often found in relational databases. ORM is a term is often overloaded, and can also refer to the suite of tools, or libraries that enable the actual mapping in a programmatic way. (freeCodeCamp 2023)
Package	Package is a term which can be interpreted in multiple ways. Package can be interpreted as multiple bundled programs or a software bundle which fulfills a certain functionality. (Rouse 2022)
Package Manager	Package Manager or a Dependency Manager is a pro-

	gram that automates the download and installation of a dependency. Dependency can be hosted in a public or a private repository. (Cox 2019)
Pull Request	Pull Request is a construct which packages contributors potentially proposed changes to a specific software projects, specific branch.
Python	Python is a high-level, actively developed, general-purpose programming language. (Python 2023)
SQA	Software Quality Assurance is a quality management process, which aims to ensure that stakeholder quality requirements will be fulfilled. For example, prevent a bug. (edvantis 2020)
SQC	Software Quality Control is the set of actions, which are derived from the SQA. For example, detect a bug. (edvantis 2020)
Technical Debt	Technical Debt is a term which refers to a situation, where speed of delivery is prioritized over perfected code, resulting software functionalities which needs to be refactored later. (ProductPlan 2023)
Transitive Dependency	Transitive dependency is a indirect dependency of a dependency, or a "dependency of dependency". (Vial 2022)
TTM	TTM, or time to market, is a definition which refers the the length of time to develop a product and release it to the market. (Carter 2023)
Unix Timestamp	Unix Timestamp is a numerical value which expresses how many seconds has passed since January 1st 1970 (UTC). (Dan’s Tools 2023)

List of Figures

List of Tables

Contents

1 Introduction

Modern software development relies heavily on software libraries, frameworks, and other reused software components. Mature and well-tested libraries are argued to increase developer productivity, improve application time-to-market, and contribute to more reliable software (Saied et al. 2018). Software reuse leads to larger codebases, increasing the attack surface for malicious actors to exploit, and might create a situation where the underlying library cannot be swapped or removed because existing functionality has been built to depend on that (Aris Papadopoulo 2020).

Opportunistic design describes a set of opportunistic practices where specific software components are reused together, which are not designed to be used together in the first place (Hartmann, Doorley, and Klemmer 2008). Opportunistic design patterns occur when inexperienced programmers copy and use code from online platforms without a complete understanding of what the code does. Typical traits of opportunistic design are that (Mikkonen and Taivalsaari 2019):

- Reused code snippets or libraries are not screened.
- A systematic, abstraction-driven approach is not followed for the sake of convenience.
- The developer does not have a complete understanding of the reused code.

This thesis studies open-source software projects developed in Go, aiming to find the implications of opportunistic design patterns. Go, a popular (StackOverflow 2023), modern and fairly new open-source programming language introduced by Google, offers a wide range of projects to study, ranging from enterprise-grade containerization solutions to smaller convenience libraries. Language enjoys a modern toolkit of different quality-of-life tools, including a package manager. Access to these tools eases software development in many stages but also creates the potential for opportunistic practices to seep into the codebases.

The thesis attempts to seek variables from the codebases that represent and contribute to the software quality and study whether these variables correlate with the usage of reused code. The thesis aims to conclude a clearer understanding of the opportunistic design patterns within software projects developed in Go.

The thesis is divided into several separate chapters according to the following structure. At the beginning, the research context, methods, and research plan are explained. The thesis initially offers a literature review of the theory of the topic, the goal of which is to ensure that the reader has enough theoretical understanding to follow the research section of the study. The research section reviews how the research plan was actually implemented, and the results section summarizes the results. Finally, the discussion and conclusion sections summarize the results and conclude the study.

2 Overview

In this chapter, the research context is reviewed, the methods used are explained, and the research plan is formulated. The research plan is used in the thesis as a framework for implementing the research.

The study observes whether software reuse contributes to less quality software and is scoped to analyze open-source projects developed in Go. The following research question was formed for the research:

'Does the Quality of Software Projects developed in Go correlate with the usage of third-party libraries?'

2.1 Research Frameworks and Methods

This section explains the theory behind the research frameworks and methods relevant to this study. Theory includes quantitative research methods, such as sections on hypothesis testing, correlation coefficients, and correlational research.

2.1.1 Quantitative Research

The quantitative research methodology was chosen because the objective of the study was to examine relationships between variables. Quantitative methodology deals mainly with numerical data, whereas the counterpart, qualitative research methodology, deals with non-numerical data (Bhandari 2022b).

2.1.2 Hypothesis and Hypothesis Testing

Hypothesis testing is a statistical research method that studies variables or the probability distribution of the studied population (Anderson, Sweeney, and Williams 2023). It is usually almost impossible or at least unnecessary to collect data on the entire population, so alternatively, a sample of the population is studied (Gravetter and Wallnau 2017).

The sample represents the population on a smaller scale, and the observations made against it are then generalized to the entire population (Gravetter and Wallnau 2017). Hypothesis testing can be divided into five steps (Henkel 1976a):

1. Selecting or formulating the hypothesis.
2. Selecting the statistical method.

3. Determine the significance level.
4. Test phase.
5. Final decisions.

The hypothesis is a statement regarding a population (Henkel 1976b). Testing the hypothesis requires two statements. The first statement is called *'Null Hypothesis'*, and denoted as H_0. H_0 represents the state where nothing has changed (Henkel 1976b). The second statement is the research statement, and is called *'Alternative Hypothesis'*, and is denoted as H_a (Henkel 1976b).

Statistical test results *test statistic* (Bevans 2022). The test statistic is a value that describes how much the relationship between studied variables from the population differs from the H_0 or from the situation where there is no change (Bevans 2022).

Test results also have a probability value (p) (Sharot November 2004). p indicates the probability of an erroneous conclusion and is also a representation of evidence against the null hypothesis.

The method for calculating p is dependent on the statistical test being used and a hypothesis being tested (Sharot November 2004).

- If the p is less than 0.05, it is statistically 'almost significant.'
- If the p is less than 0.01, it is statistically 'significant.'
- If the p is less than 0.001, it is statistically 'very significant.'

Significance level, α, is determined by researcher (Henkel 1976b). The significance level explains the threshold for the possibility where the null hypothesis would be rejected and is usually either 0.01 or 0.05 (Henkel 1976b). In other words, if the calculated probability value is below, for example, 0.01, it would mean that there is over 99.9 percent probability that this result has not occurred by chance.

Statistical test variables, α, and p, help to determine whether the null hypothesis is going to be rejected or not (Henkel 1976b). In other words, whether the null hypothesis will be accepted or rejected depends on how it compares to α (PennState Eberly College of Science 2023):

- $p <= \alpha$, then it is unlikely that the 'null hypothesis' is true, and it can be rejected.
- $p > \alpha$, then it is likely that the 'null hypothesis' is true and can be accepted.

Deciding which hypothesis to accept or reject can lead to two different errors (Henkel 1976a). *Type I Error*, rejection error, when the null hypothesis is rejected where it should have been accepted or *Type II Error*, acceptance error, when the null hypothesis is accepted where it should have been rejected (Henkel 1976a).

After the testing phase, the execution of the testing must be critically examined. Criticism can be split into three different groups: technical, philosophical, and practical. (Tietoarkisto 2023)

Technical criticism is valid when tests are based on certain assumptions. Philosophical criticism can vary a lot more. The most important criticism could be the character of the null hypothesis. Usually, it is assumed that some parameter values are zero, but it is actually unlikely that any of the parameter values would actually be zero. (Tietoarkisto 2023)

Practical criticism can be split into two different categories. The first category is the misuse of statistical tests. The researcher is not completely aware of the research methods, is not choosing the appropriate test for the research problem, or is unable to correctly interpret the research results. The second category is where a researcher overly emphasizes the result of the statistical tests instead of examining the actual results. (Tietoarkisto 2023)

2.1.3 Correlational Research and Correlation Coefficient

Correlational research as a research design investigates relationships between variables. Correlation is a measure of the relationship between two variables. Correlation has a strength and a direction, which can be either positive or negative. (Bhandari 2022a)

Correlational research is suitable for researching causal and non-causal relationships. Causality explains a scenario where there is cause and effect relationship between two variables. A linear relationship between variables is a form of a causal relationship. One variable increases, and another variable increases or decreases in proportion. A non-causal relationship between two variables is a relationship where one does not directly cause another, but the relationship is a correlation or coincidence. (Bhandari 2022a)

Correlation is represented as a correlation coefficient (Bhandari 2022a). The correlation coefficient has a strength and a direction; 1 represents perfect correlation; when one variable increases, the other variable increases proportionally. 0 represents a situation with no relationship between variables, and -1 represents perfect negative correlation (Bhandari 2022a). Correlation coefficients have requirements and are sensitive to outliers in the data (Statology 2019). The coefficient can be misleading if the requirements are not met (Statology 2019).

Pearson’s correlation coefficient (denoted as r) is a parametric test and is used to measure the correlation between variables with linear relationships (Bhandari 2021). The coefficient has the following requirements for the data (Turney 2022):

- Measured variables must be quantitative.
- Measured variables must be normally distributed.
- Measured data should not have extreme outliers.
- Measured variables must have a linear relationship.

Spearman's rank correlation coefficient (denoted as ρ) is a non-parametric test and is used to measure the correlation between variables with monotonic relationships. The coefficient assesses relationships based on the ranks instead of the actual values. The coefficient has the following requirements for data (Bhandari 2021):

- Measured variables must be ordinal, interval, or ratio.
- Measured variables must have a monotonic relationship.
- Measured data should not have extreme outliers.
- Sample size should generally be higher.

Kendall's rank correlation coefficient (denoted as τ) is also a non-parametric test and is used to measure the correlation between variables with monotonic relationships. The coefficient assesses relationships based on pairs of observations, and the strength of the relationship is measured from the concordance and discordance between pairs. (Magiya 2023):

- A Concordant pair is a pair of observations where the ranks of both observations are in the same direction.
- A Discordant pair is a pair of observations where the ranks of both observations are in the opposite direction.

Kendall's coefficient is generally preferred for smaller sample sizes of data compared to Spearman's method. The method has the following requirements for the data (Magiya 2023):

- Measured variables must be ordinal, interval, or ratio.
- Measured variables must have a monotonic relationship.
- Measured data should not have extreme outliers.
- Sample size should generally be smaller.

The most relevant correlation coefficients for this study are Pearson, Spearman, and Kendall. There are also other correlation coefficients that, although useful in other contexts, are beyond the scope of this study.

2.2 Research Design

This section designs and introduces a research plan for the study. The research plan is ultimately based on "Explanatory Data Analysis" and utilizes "Hypothesis Testing" as a research framework. Exploratory Data Analysis (EDA) describes a sequence of steps to obtain insights from the data. EDA has an emphasis on visualizing the data and can be split into multiple steps (Madhugiri 2023):

- Interesting variables contributing to the studied phenomenon must be identified.
- Data must be collected and preprocessed for analysis.
- Variables and their relationships have to be studied.
- Correct statistical methods have to be identified and applied.
- Results have to be visualized and analyzed.

2.2.1 Define Hypothesis Testing Layout

Research question is split into null hypothesis, H_0, and alternative hypothesis, H_a, in a following manner:

H_0: Utilizing third-party libraries in software projects developed in 'Go' does not affect the software project's quality.

H_a: Utilizing third-party libraries in software projects written in 'Go' affects the software project's quality.

2.2.2 Identify Variables

What exactly is going to be collected is determined by identifying relevant variables that are of interest. The objective is to understand the relationships between third-party code usage and quality in software projects, primarily dictating what variables are valuable for the study. The variables available mainly relate to the metadata of the version-controlled repositories or are simple derivatives that can be locally computed.

2.2.3 Collect, Preprocess and Standardize the Dataset

The sample is collected programmatically through public APIs of online code hosting platforms, such as GitHub (GitHub 2023a). The sample consists of established open-source repositories developed in Go. Identified variables are extracted from the sample. Variables are stored in persistent storage and are preprocessed by removing broken entries from the data and standardized by normalizing values to the same range. Outliers in the dataset are going to be addressed. The variables then are used to form composed variables that aim to represent the quality of the software project. Normalization is done by utilizing the min-max normalization, with the following formula presented in figure (1) (Ciaburro, Ayyadevara, and Perrier 2018).

$$q_x = \frac{x - \min(v)}{\max(v) - \min(v)} \tag{2.1}$$

Figure 1: Formula: Min-Max Normalization

2.2.4 Determine and Apply Statistical Tests

The study seeks to understand the relationship between different variables. This supports the selection of the correlation coefficient as a statistical tool (Bhandari 2021). Variables have to be further studied before the specific correlation coefficient can be chosen because correlation coefficients have different requirements for the input data (Bhandari 2021).

The coefficient also requires a determined significance level, α, α of 0.05 is chosen for the study. Tests are executed programmatically with Python, which is a general-purpose programming language with a large ecosystem for scientific computing (Cornell University 2010).

2.2.5 Analysis

Statistical tests result in correlation coefficients, probability values, and other material, such as visualized distributions and correlation matrices from and between the studied variables.

The research question is studied, and conclusions are drawn from the results of hypothesis testing. Other results are also studied, reflected, and further analyzed since they might uncover underlying trends and patterns, which were not in scope for this study but can be studied in later studies.

2.2.6 Discussion and Conclusion

The discussion section reflects the meaning, importance, and relevance of the results and compares the findings to the literature review, which is carried out in the theory section of the study (McCombes 2022). The discussion also reflects on the practicality of the results in relation to the statistical significance and considers the limitations of the study.

The conclusions summarize the research, the most important findings, and how the research question was answered. Future research topics are identified from the limitations of this study and possible unanswered questions. (George and McCombes 2022)

3 Theory

This chapter is a literature review. The literature review aims to ensure that the reader has enough theoretical understanding to understand and critically analyze the research section of the study. (McCombes 2023)

3.1 Software Quality

Software quality measures how well a software product establishes a set of requirements. Requirements, determined by the stakeholders, aim to generate stakeholder satisfaction. Software is considered to be high quality when it fulfills all the requirements. Software quality ignores the development paradigm, which means that requirements can be defined before or iteratively during the development process. (Galin 2018a)

Software errors can be a measure of software quality and can be defined as the inability to meet the stakeholder's requirements (Asq 2023). Programmers or designers can introduce errors that can be grammatical, logical, or, for example, design errors (Galin 2018b). Software fault, on the other hand, is a software error that leads to the malfunctioning of certain applications of the software (Galin 2018b). All errors are not faults, and usually, only the software faults that disrupt the use of software are of interest (Galin 2018b).

3.1.1 Software Quality Assurance

Software quality control (SQC) is a set of activities with the objective of evaluating the quality of a final software product and ensuring that the shipped product has an acceptable quality level. Software quality assurance (SQA), on the other hand, is a set of activities performed throughout the development life cycle in order to detect and prevent software errors and ensure that the quality is at an acceptable level. (Galin 2018c)

Software quality engineering adopts practices from software quality assurance and software quality control (Galin 2018d). Software fault management is a quality control approach that utilizes counting and categorizing defects by their severity. Number of defects is a way to measure software product quality (Asq 2023).

Attributes in the requirements can be categorized as "Software Quality Factors" (Galin 2018e). These factors build models that are used as a method to classify software quality requirements (Galin 2018f).

3.1.2 McCall's Quality Model

McCall's factor model classifies all Software Requirements into 11 quality factors, which can be classified into three categories: operation, revision, and transition factors (Galin 2018f). We start by listing operation factors, which are the following (Galin 2018g):

- Correctness: Correctness requirements refer to the required accuracy, completeness, and up-to-datedness of the output values.
- Reliability: Reliability requirements refer to the software system's ability to provide functionality regardless of failures.
- Efficiency: Efficiency requirements refer to the resource requirements of the software system so the system can guarantee service level and at the same time meet other requirements.
- Integrity: Integrity requirements refer to the security requirements.
- Usability: Usability requirements refer to the resources needed to train someone from the ground up to operate the software system.

Revision factors are the following (Galin 2018g):

- Maintainability: Maintainability requirements refer to the user's ability to identify and correct a failure and verify the success of the correction.
- Flexibility: Flexibility requirements refer to the capability and effort required to support maintenance activities.
- Testability: Testability requirements refer to the testing process of a software system.

Transition factors are the following (Galin 2018g):

- Portability: Portability requirements refer to the software system's ability to adapt to different environments with different hardware and operating systems.
- Reusability: Reusability requirements refer to the ability to reuse complete software systems or components in new software systems.
- Interoperability: Interoperability requirements refer to the ability to create interfaces towards other software systems.

3.1.3 ISO/IEC 25010 Model

"ISO/IEC 25010:2011" model is developed by a joint ISO/IEC international professional team. (Galin 2018h) ISO/IEC 25010:2011 outlines eight quality characteristics, which are the following (Galin 2018h):

- Functional Suitability: Functional suitability refers to the software system's

ability to perform the functions required by the end-user.
- Performance Efficiency: Performance efficiency refers to the software systems hardware resource requirements, and it is a correlation to complete the software system tasks. The lower the requirements, the higher the performance.
- Compatibility: Compatibility refers to the software system or a smaller software component's ability to exchange information with other systems or components and in conjunction to perform other functions while interoperating with different hardware and software configurations.
- Security: Security refers to the software system's ability to protect the system, data stores, and information produced from unauthorized access.
- Usability: Usability is like in McCall's model.
- Reliability: Reliability is like in McCall's model.
- Maintainability: Maintainability is like in McCall's model.
- Portability: Portability is like in McCall's model.

3.1.4 Alternative Models

Alternative models propose more quality factors, which are not incorporated in McCall's or ISO/IEC 25010 Models (Galin 2018i). Additional factors, compared to McCall's model, are the following (Galin 2018i):

- Effectiveness: Effectiveness refers to the ability to successfully complete software development tasks, accounting for the schedule and error frequency.
- Evolvability: Evolvability refers to the software system's ability and effort to support future requirements, technologies, and changes in the operating environment.
- Expandability: Expandability refers to the software system's ability to scale to a wider end-user population.
- Extensibility: Extensibility refers to the software system's ability and effort to support future requirements that result from economic and technological developments.
- Human Engineering: Human Engineering refers to the software systems interfaces towards the end-user and their ease of use.
- Manageability: Manageability refers to the software systems requirements for administrative tooling. Administrative tooling could, for example, enable software modification during the software development and maintenance periods.
- Modifiability: Modifiability refers to the effort that goes into modifying software systems against specific requirements of customers.
- Productivity: Productivity refers to the software system's speed and how

fast the software system can complete tasks.
- Safety: Safety refers to the software systems requirements that prevent conditions that might be hazardous to the equipment and the people operating the equipment.
- Satisfaction: Satisfaction refers to the software system's ability to fulfill the expectations of the end-user.
- Supportability: Supportability refers to the ease of performing maintenance tasks to the software system.
- Survivability: Survivability refers to the software system's continuity of service.
- Understandability: Understandability refers to the user's ability to understand how to operate the software systems.
- Verifiability: Verifiability refers to the requirements that enable verifying the design and features of the software systems.

3.2 Software Reuse

Software reuse is a practice where systems, applications, software components, or directly the code is reused in a different project where it was originally written. This was more of an uncommon practice until the 2000s, but it is nowadays extensively used to build new software systems. (Sommerville 2015)

Software reuse is an efficiency measure. Internet and the wide adoption of open-source have increased software reuse to the degree that practically every software component today relies at least partly on reused code. Reusable code can be shared in different ways. For example, the wide availability of different package managers and other development tools enables software developers to pull shared and prepackaged code with ease and incorporate that into the workflow. (Vial 2022)

Multiple sources argue, that software reuse clearly has the benefits. Arguments includes the following claims (Pandey 2022, Sommerville 2015):

- Faster development time, which contributes to faster time-to-market value.
- Lower overall development costs, in general, more effective use of software developer's time.
- More predictable development costs because part of the software is already written.
- Reused software might be more robust because it is more "battle-tested."

On the contrary, multiple sources also argue that software reuse clearly has its pitfalls. These arguments includes the following claims (Vial 2022, Sommerville 2015):

- If software is created to be reusable, that involves creation and maintenance costs.
- Studying existing software artifacts might require a significant time investment.
- Deprecated reused software artifacts might lead to increased costs due to self-writing in the first place.
- Third-party software might have little to no support network.
- Side effects in the code.
- Security vulnerabilities in the code.

Although software reuse is easy with modern tools, it is not always effective. Here are Charles W. Krueger's Four Truisms for Effective Software Reuse (Krueger 1992):

- *"For a software reuse technique to be effective, it must reduce the cognitive distance between the initial concept of a system and its final executable implementation."*
- *"For a software reuse technique to be effective it must be easier to reuse the artifacts than it is to develop the software from scratch."*
- *"To select an artifact for reuse, you must know what it does."*
- *"To reuse a software artifact effectively, you must be able to "find it" faster than you could 'build it.'"*

3.2.1 Opportunistic Software Reuse

Software reuse refers to a practice where software systems are developed by combining existing software and components (Mäkitalo et al. 2020). Opportunistic software reuse, on the other hand, refers to a practice where software reuse is done with components and software which was not designed to be used together in the first place (Mäkitalo et al. 2020).

Practice endorses convenience and productivity in the short term. Developers grab code snippets here and modules there and cook them together into a working piece of software (Mäkitalo et al. 2020). Visible code, however, is not actually the whole story, as reused code can be many times more than the actually self-written code (Mäkitalo et al. 2020).

Opportunistic reuse has been seen as a troublesome phenomenon because hidden code (in the form of transitive dependencies) and the fact that the reused assets are not designed to be used together in the first place could lead to unknown side effects. Opportunistic reuse is only ticking 1 or 2 boxes from the Kruger's truisms. Development for sure is easier, but not having knowledge of the reused asset, a resulting system might be unnecessarily complex, and the developer might not even be able to build the software from the ground-up to compare whether a solution with reused assets is even faster.

3.2.2 Unwanted Consequences of Software Reuse

The industry has shifted to a reputation-based recommendation system for software that is being reused. Unwanted consequences of reusing software depend on the tolerated risk level. The level might vary. For example, a developer's own hobby project will probably tolerate more risk than an enterprise project that has paying customers. (Cox 2019)

Unwanted consequences can be mitigated, for example, by having a review process in place. The review process could include the following steps (Cox 2019):

- Inspect Dependency Documentation: The developer needs to understand how to interact with the library. Is there documentation, and is it high quality enough that it can be worked with?
- Issue Code Review: Sometimes, the developer needs to be able to debug the library code. It is good practice to inspect the code before to determine whether the developer can debug it.
- Determine Support Level: Determine whether the library has an active support network.
- Activity: Determine whether the library is actively developed.
- Popularity: Check whether others rely on this library.
- Vulnerability Trend: Has there been a trend of vulnerabilities with this library?
- License: What kind of licensing does this library have?
- Transitive Dependencies: Does the library have a lot of transitive dependencies?

The library might fail the screening process later, even if it passes now. Future needs to be planned as well, and it is a good idea to monitor the used libraries in a frequent manner. (Cox 2019)

4 Research

This chapter walks through how the research design was executed. The chapter explains the process of variable identification, dataset collection, dataset preprocessing, dataset postprocessing, and dataset analysis.

4.1 Identifying Dataset Content

When the objective was to measure quality in a meaningful way, in this context, it practically meant identifying variables from the project that are inversely proportional to the undesirable consequences of software reuse. When the source of data was version-controlled software repositories, these variables were available programmatically in numerical format:

- Issues (Open or Closed): Issues are collections of data that are used to track development activity (GitHub 2023b).
- Commits: Commit is a construct that represents a record of changes to files. Commits have a unique identifier (GitHub 2023c).
- Stargazers: Stargazers or Stars are a popularity metric in GitHub. The metric represents the number of GitHub users that have "starred" certain repositories (Md. Fahim Bin Amin 2023).
- Forks: Fork is a repository that is essentially a copy from the original "upstream" repository (GitHub 2023d).
- Pull Requests (Open or Closed): Pull Requests are a collection of changes that a developer proposes to a certain branch in a certain repository in GitHub (GitHub 2023e).
- Releases: A release is a structure that allows capturing, tagging, and publishing a specific state of code on GitHub. The release can include relevant files and a release note (GitHub 2023f).
- Network Events: Network Events or Events is a metric that represents the numerical value of all activity in a certain GitHub repository (GitHub 2023g).
- Subscribers: Subscriber is a GitHub user that has subscribed to a certain scope of events from a certain GitHub repository (GitHub 2023h).
- Contributors: A contributor is a GitHub user who has contributed changes to a certain repository (GitHub 2023i).
- Watchers: Watcher is a GitHub user who has subscribed to all events from a certain GitHub repository (Metrics Toolkit 2023).

Dates could also be used as numerical values by converting them to Unix timestamps. The following date variables were available programmatically:

- Latest Release.
- Creation Date.

Variables to measure the size and proportion of third-party or library code in software projects were not directly available programmatically. The following variables had to be computed:

- Self-Written Lines of Code.
- Library Lines of Code.

Variables "Self-Written Lines of Code" and "Library Lines of Code" were also used to compute variables "Library Lines of Code Proportion" and "Self-Written Lines of Code Proportion." Variables represent the proportion of certain types of code within the codebase.

Composite variables were formed from variables that did not represent lines of code. Composite variables were formed by grouping similar variables together. These variables tend to be inversely related to the undesirable consequences of software reuse:

- Activity: Open Issues, Closed Issues, Commits, Open Pull Requests, Closed Pull Requests, Network Events, Contributors.
- Maturity: Creation Date, Latest Release, Releases.
- Popularity: Stargazers, Forks, Subscribers, Watchers.

Activity in this context means regular activities in the software project, such as regular development activity, updates, discussion, and bug fixes. Active software projects and libraries promote:

- Security: When the project receives regular updates and bug fixes, it makes the project less likely to have security vulnerabilities. If vulnerabilities are detected, they are quickly fixed. (Cemazar 2022)
- Compatibility: When the project receives regular updates and bug fixes, the project is more likely to be compatible with newer versions of software frameworks, operating systems, and programming languages. However, proprietary hardware does not always support open-source software like drivers. (Cemazar 2022)
- Support: Active projects could also promote an active community of developers who can provide support.
- Features: Active projects could indicate regular updates with new features and improvements.

Maturity in this context means the software project is stable, documented, and supported. Maturity promotes stability while using the software in different environments. (Pronschinske 2016) These qualities can contribute to scalability and maintainability in the long run:

- Stability: Mature projects can be more tested and used in production environments.
- Documentation: Mature projects can have more comprehensive documentation, tutorials, and examples of utilizing them and troubleshooting issues.

Popularity in this context means other projects and enterprises use and endorse the software project. Popularity can contribute to the following qualities:

- Reliability: Popularity can indicate that the software project has been tested and used by many software engineers, which promotes reliability and stability.
- Innovation: Popular libraries are more likely to be updated with new features and improvements.

4.2 Collection of the Sample

Variables that combine the dataset were collected with a self-developed research tool (referenced as a *collection tool* in this study) that leverages "Sourcegraph" (Sourcegraph 2023), a code intelligence platform that indexes publicly available open-source repositories (such as repositories hosted in GitHub), as a data source. Sourcegraph was selected as a data source because it caches the metadata of queried repositories, offers powerful GraphQL API, and has comprehensive documentation. In short, it is fast and easy to use.

The collection tool is responsible for collecting the data set as shown in figure 2. First, a request is sent to the collection tool via the REST API; after this, the Collection Tool starts collecting data. Data is retrieved from Sourcegraph's GraphQL API and the retrieved entries are then stored in the database. After all the entries have been retrieved, each entry in the database is processed locally by calculating the missing variables, such as the number of library code and the proportion of library code, and then the processed entry is updated to the database.

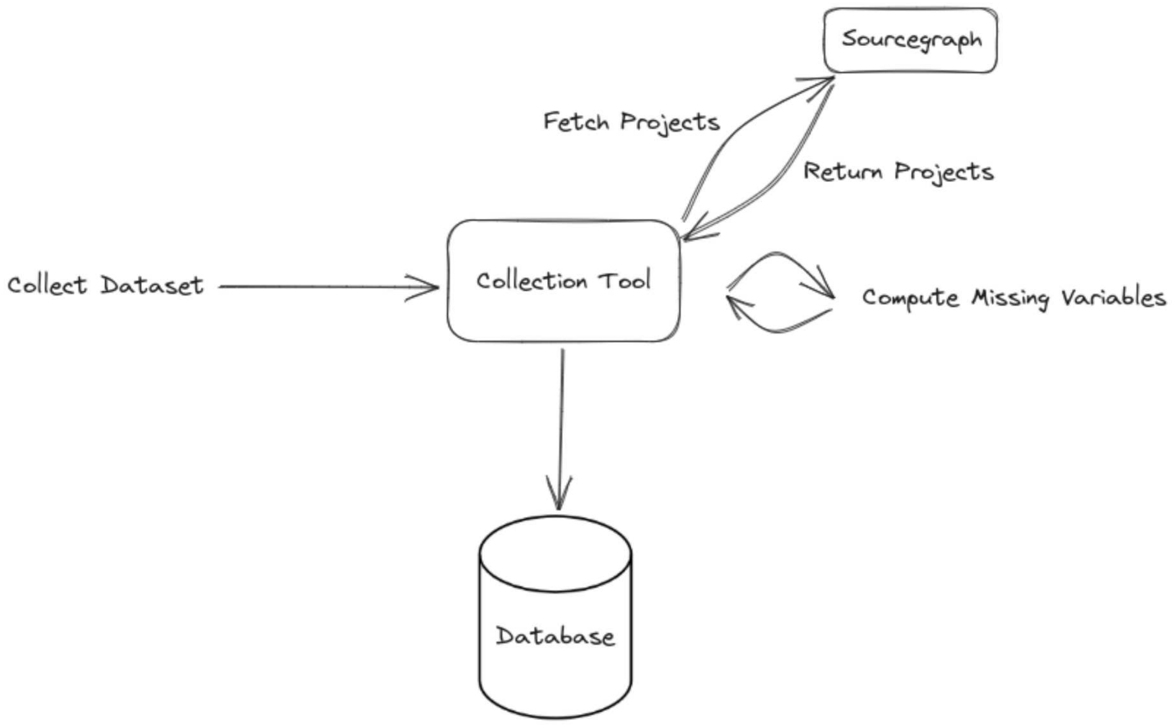


Figure 2: Data Collection Procedure

4.2.1 Collection Tool

This subsection introduces how the collection tool was developed. The tool was developed in Go and is publicly viewable on GitHub (GitHub 2023k). The tool is essentially a Web API, which offers functionality to query the remote data source for open-source project metadata, fetch defined variables, compute the missing defined variables, and store the data in the database for further processing.

The collection tool utilizes concurrency due to expensive operations included in the dependency processing, which is required for repository size calculation. Dependency processing includes downloading files, reading, writing, and deleting data from the local file system. Processing is designed using a design pattern where processes are labeled either as "Consumer" (C) or "Producer" (P), depending on the role of the process (Cornell University 2010). P and C are both implemented in the tool as goroutines, which are essentially lightweight processor threads; thus, they run concurrently (Go 2023).

P is responsible for downloading dependencies and then signaling C through a channel, which also acts as a queue, that a job is waiting to be processed. C is then responsible for postprocessing the jobs, including calculating downloaded library sizes, caching, and saving the results. Calculation is done by leveraging *"gocloc"* library (GitHub 2023l).

P and *C* goroutines are allowed to spawn more worker goroutines, but the total amount of workers is safeguarded with a semaphore. In this case, Semaphore is a programmatic safeguard implementation, which blocks more worker goroutines spawning than available CPU cores. Semaphore implementation is not always like what is described here; the CPU most likely can handle more goroutines than allowed in this implementation.

The collection tool uses a relational database, PostgreSQL, to store the data for further processing. Database connection and interaction within the collection tool are handled with an ORM library, and the exchange itself has been implemented as CRUD operations. In short, the collection tool has APIs to create, read, update, and delete database entries.

4.3 Preprocessing the Dataset

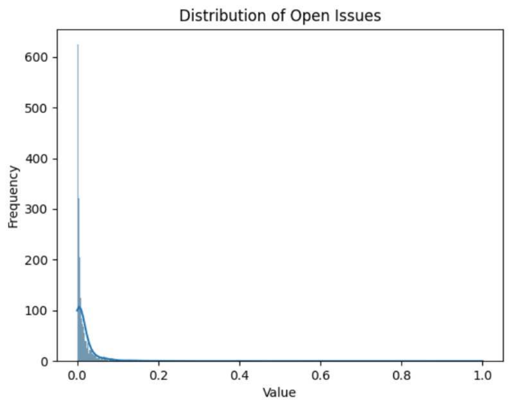
Preprocessing and postprocessing steps were conducted programmatically using self-written *"analysis tool"*, a lengthy Python script. Python was selected for processing and analysis because it is straightforward and offers many tools for data analysis (MOOC.fi 2023). The analysis tool is publicly viewable on GitHub (GitHub 2023j). One preprocessing step is cleaning the dataset to allow programmatic analysis later on. The following requirements were defined for the cleaning, keeping the set size moderate and allowing further processing of the set:

- Entries with missing fields, or in other words, "broken entries," are removed from the dataset.
- Entries with less than 100 stargazers are removed from the dataset.

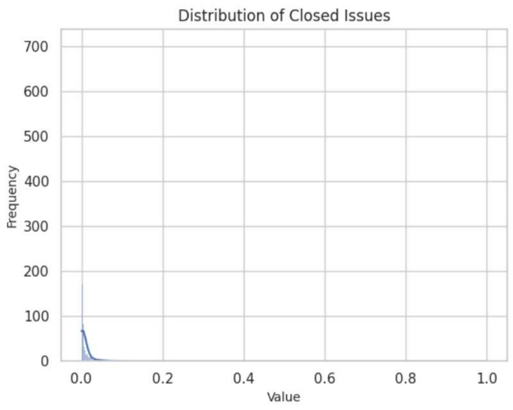
The analysis tool reads entries from the database, converts the date variables to unix timestamps and then normalizes all the variables to a standardized range and then saves them back to the database. Cleaned dataset, the sample, consists of approximately 2000 open-source software projects developed in Go that are hosted in GitHub and have more than 100 stargazers.

4.4 Postprocessing the Dataset

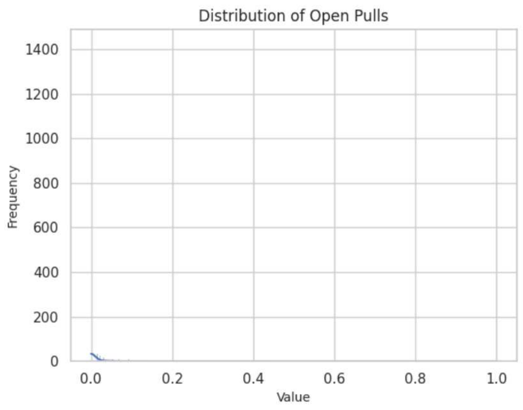
Variables were first visualized as distributions, which were used to define what statistical test was suitable for the analysis. Figures 3, 4 and 5 uncover that variables were not distributed normally.



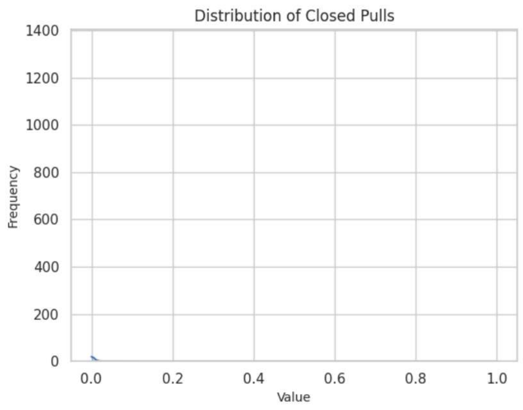
(a) Open Issues



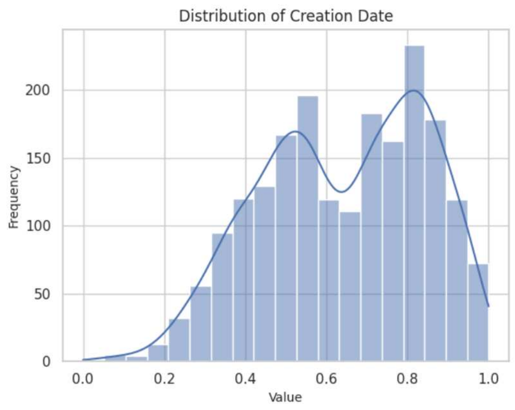
(b) Closed Issues



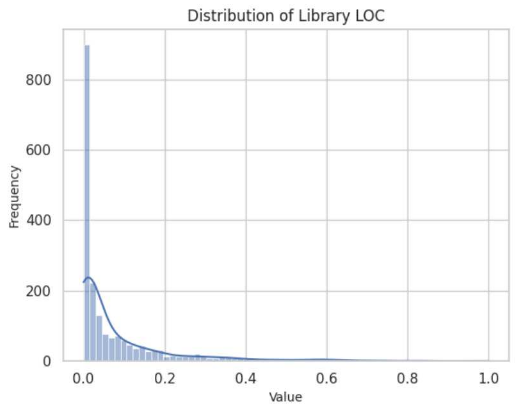
(c) Open Pulls



(d) Closed Pulls

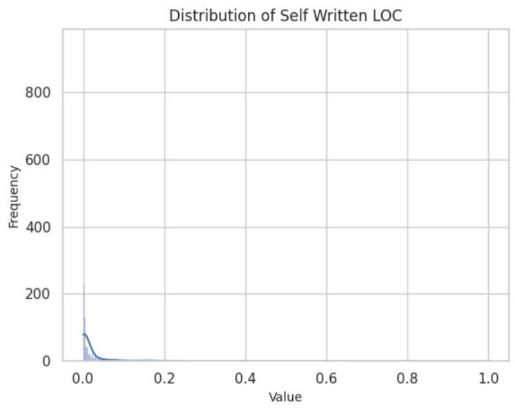


(e) Creation Date

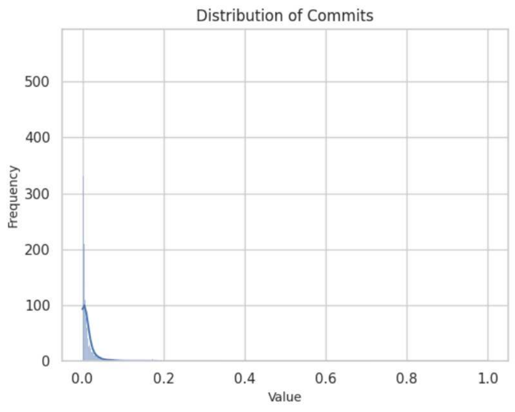


(f) Library LOC

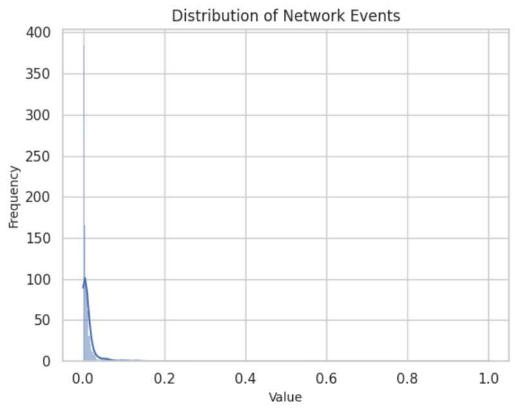
Figure 3: Distributions (Part 1)



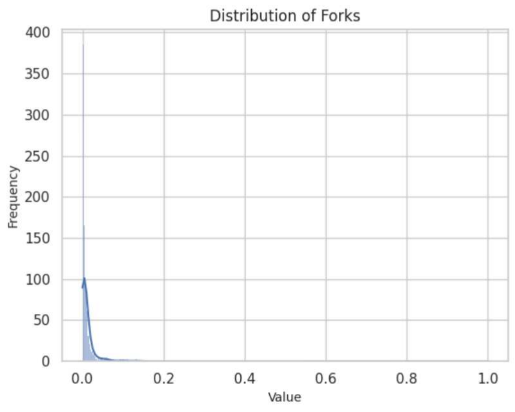
(a) Self-Written LOC



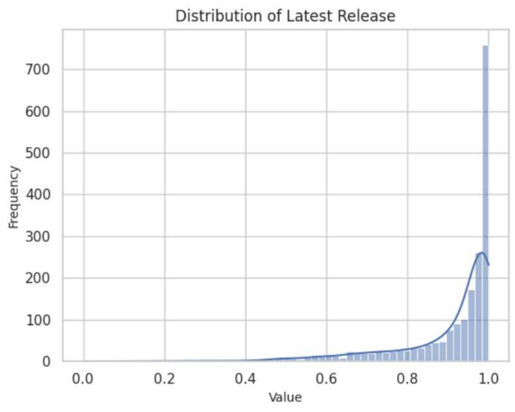
(b) Commits



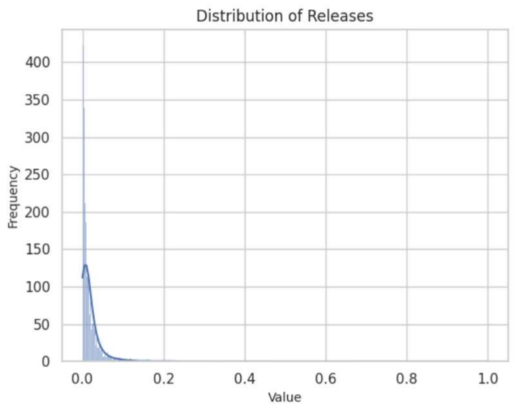
(c) Events



(d) Forks

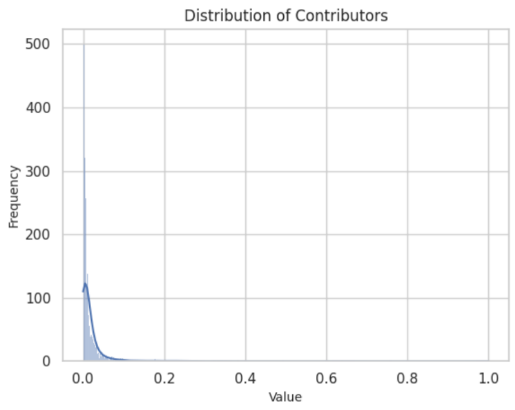


(e) Latest Release

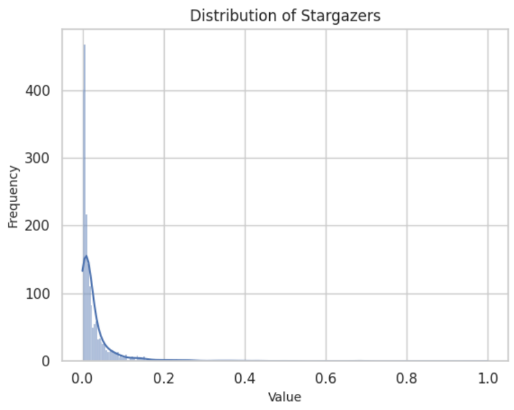


(f) Releases

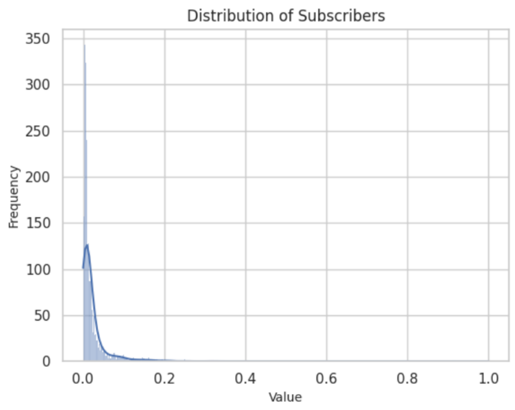
Figure 4: Distributions (Part 2)



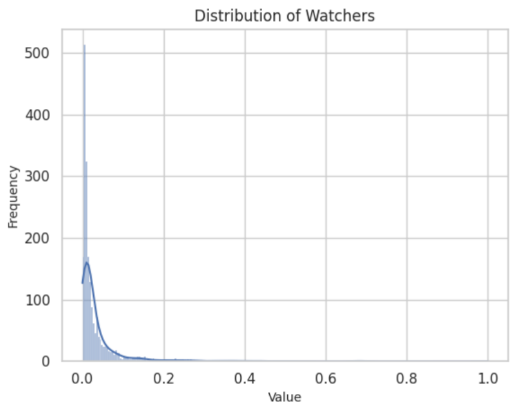
(a) Contributors



(b) Stargazers



(c) Subscribers



(d) Watchers

Figure 5: Distributions (Part 3)

When the objective was to study the relationships between variables, for example, statistical methods to analyze linear relationships was unsuitable for this dataset. However, spearman's rank correlation coefficient was suitable and was selected as the statistical test because the dataset fulfills the requirements for the method.

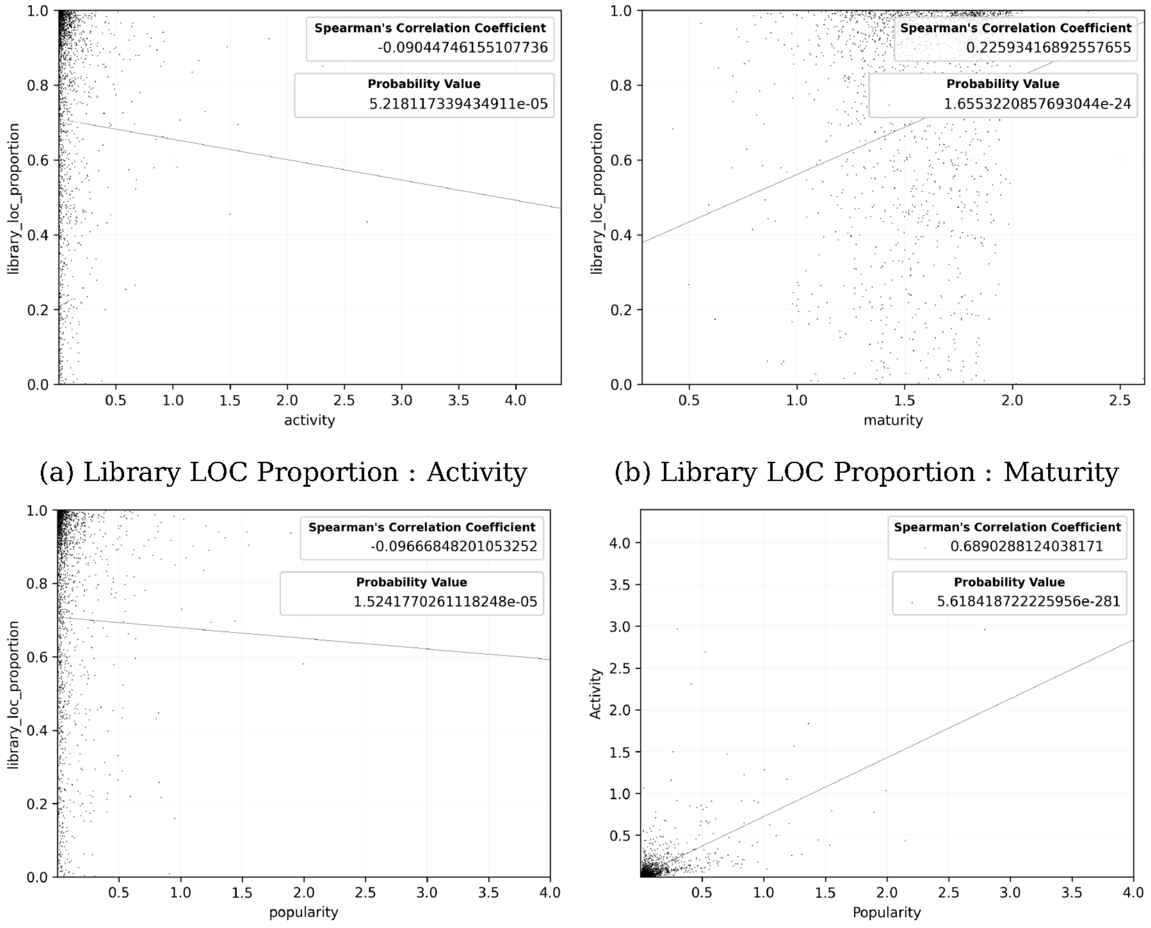
- Sample size was substantially high.
- Data has no extreme outliers.
- Relationships between variables are monotonic.

Variables were then themed, and then used to form composite variables, which are a weighted sum of underlying variables where every variable has equal weight.

- Popularity: Weighted sum of stargazers, forks, subscribers and watchers.

- Activity: Weighted sum of open issues, closed issues, commits, open requests, closed requests, network events and contributors.
- Maturity: Weighted sum of creation dates, latests releases and count of releases.

The relationships between composite variables and different LOC variables were visualized as plots, see figures 6, 7 and 8, for further analysis. Plots contain and represent the relevant data. However, representing data this way is very noisy, and the sheer amount of plots makes it difficult to analyze.



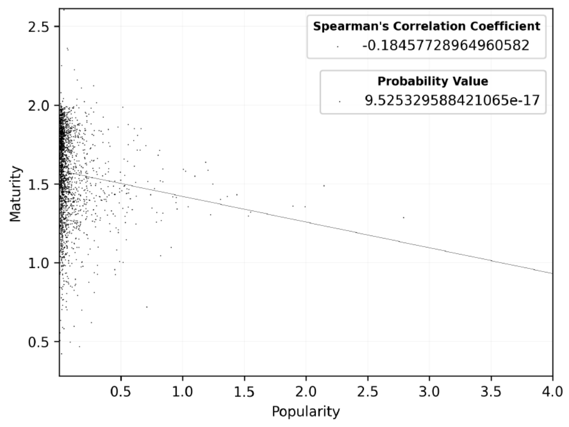
(a) Library LOC Proportion : Activity

(b) Library LOC Proportion : Maturity

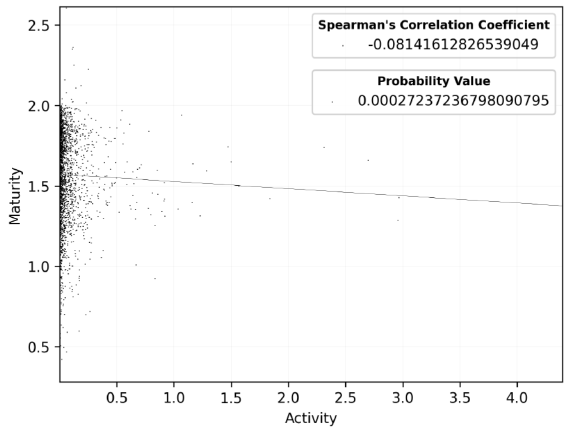
(c) Library LOC Proportion : Popularity

(d) Popularity : Activity

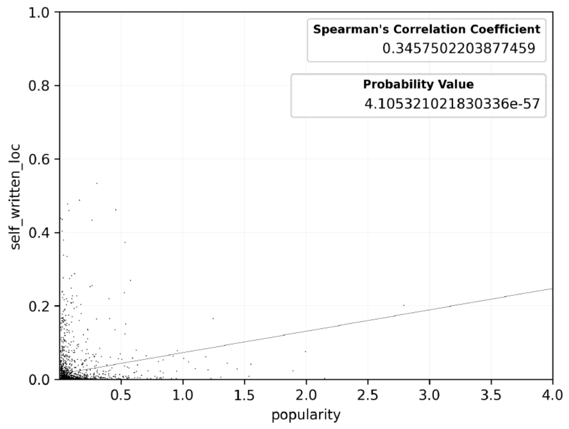
Figure 6: Plots (Part 1)



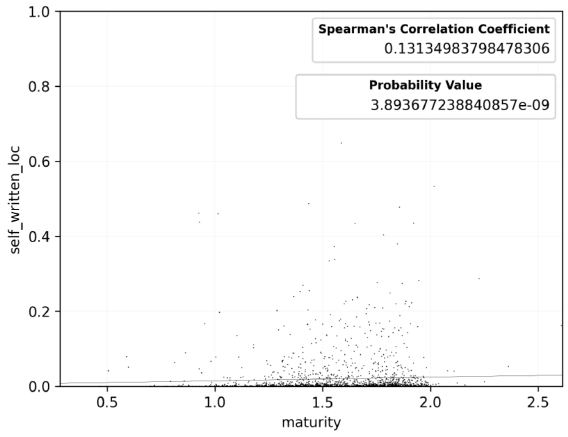
(a) Maturity : Popularity



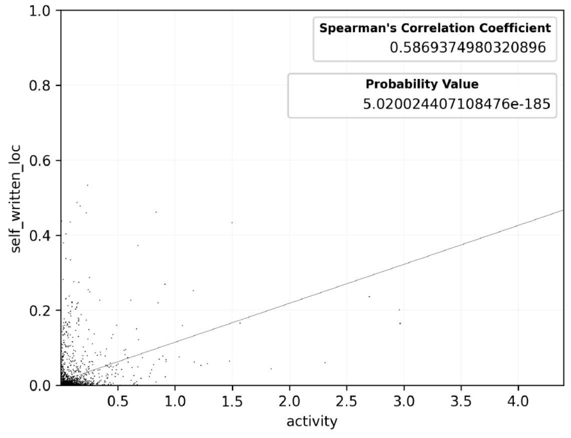
(b) Activity : Maturity



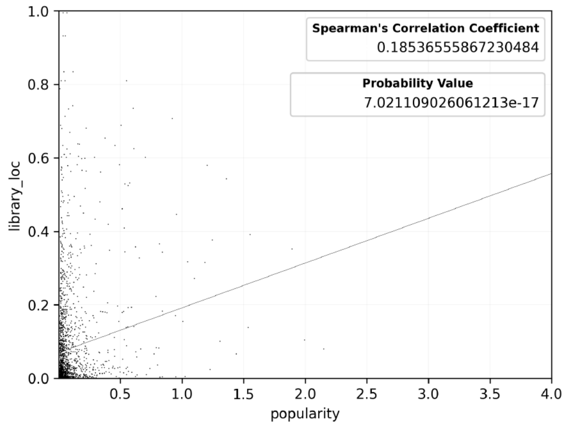
(c) Popularity : Self-Written LOC



(d) Maturity : Self-Written LOC

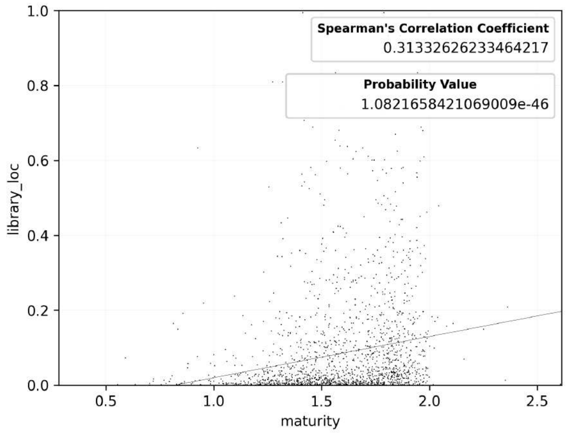


(e) Activity : Self-Written LOC

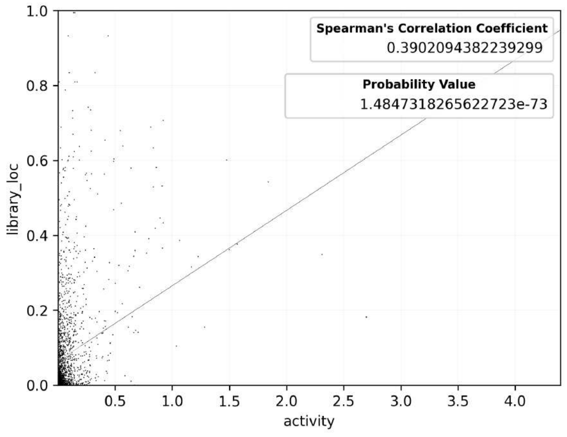


(f) Popularity : Library LOC

Figure 7: Plots (Part 2)



(a) Maturity : Library LOC



(b) Activity : Library LOC

Figure 8: Plots (Part 3)

Relevant information had to be presented in a smaller space. Correlation coefficients were the most interesting data, so a correlation matrix was chosen as the form of data presentation. The matrix was a more useful way to extract more meaningful information from the variables than the originally captured graphs, and could be easily visualized as a heatmap, see figure 9. However, the graphs can be used as references for the results.

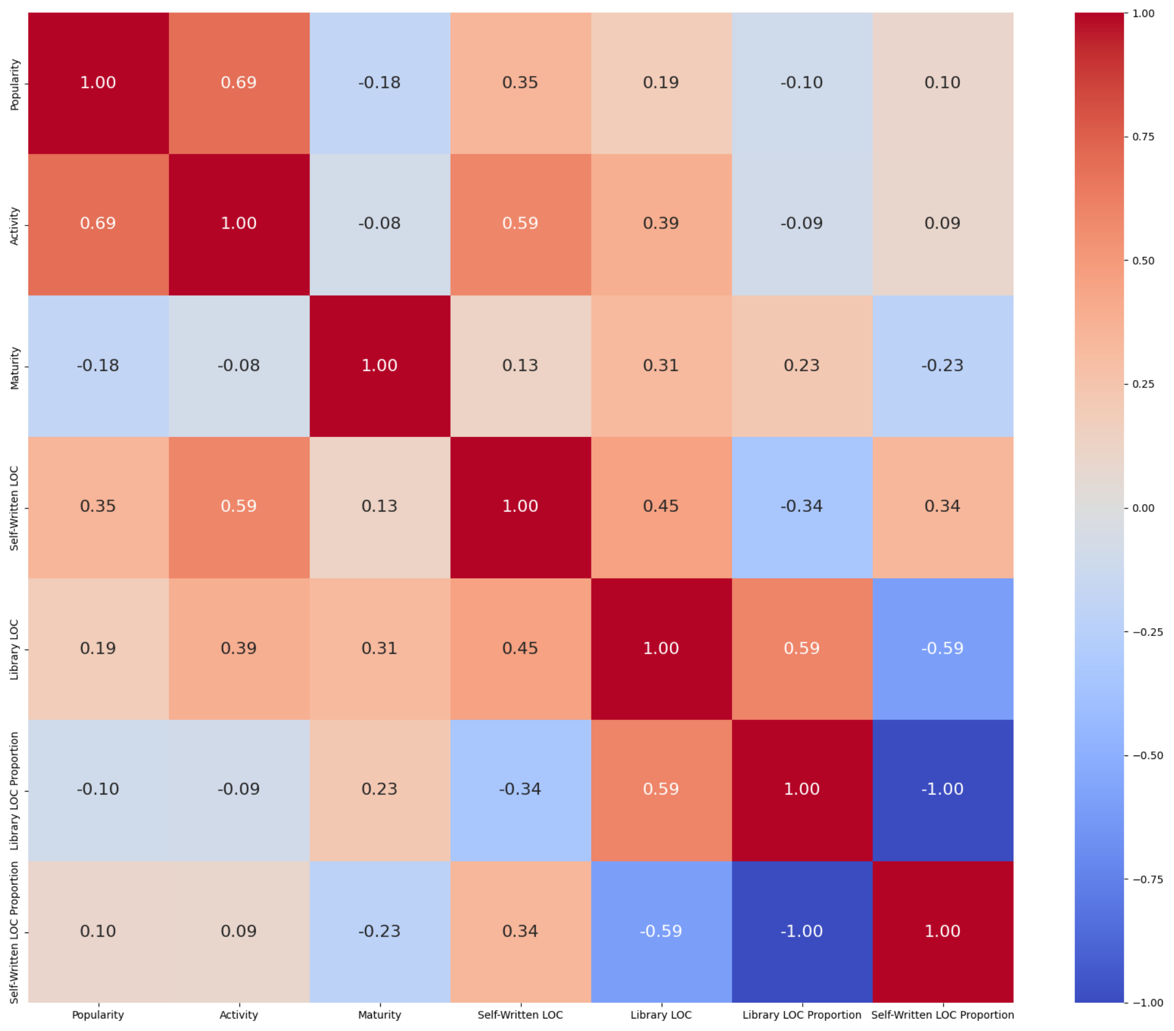


Figure 9: Spearman’s Correlation Coefficients Heatmap

5 Results

This chapter presents the analyzed results of the study. Results are presented in the table 1 and they are interpreted with the following self-defined correlation strength thresholds:

- -1.0 to -0.7: Very Strong Negative Correlation
- -0.7 to -0.5: Strong Negative Correlation
- -0.5 to -0.3: Moderate Negative Correlation
- -0.3 to -0.1: Weak Negative Correlation
- -0.1 to 0.1: Very Weak or No Correlation
- 0.1 to 0.3: Weak Positive Correlation
- 0.3 to 0.5: Moderate Positive Correlation
- 0.5 to 0.7: Strong Positive Correlation
- 0.7 to 1.0: Very Strong Positive Correlation

Variables	ρ	Correlation	p-value	Diagram
Library LOC Prop. : Activity	-0.09	Very Weak Neg.	5.22e-05	6a
Library LOC Prop. : Maturity	0.23	Weak Pos.	1.66e-24	6b
Library LOC Prop. : Popularity	-0.10	Weak Neg.	1.52e-05	6c
Popularity : Activity	0.69	Strong Pos.	5.62e-281	6d
Maturity : Popularity	-0.18	Weak Neg.	9.53e-17	7a
Activity : Maturity	-0.08	None	0.0003	7b
Popularity : Self-Written LOC	0.35	Moderate Pos.	4.11e-57	7c
Maturity : Self-Written LOC	0.13	Weak Pos.	3.89e-09	7d
Activity : Self-Written LOC	0.59	Strong Pos.	5.02e-185	7e
Popularity : Library LOC	0.19	Weak Pos.	7.02e-17	7f
Maturity : Library LOC	0.31	Moderate Pos.	1.08e-46	8a
Activity : Library LOC	0.39	Weak Pos.	1.48e-73	8b

Table 1: Correlation Coefficients, p-values, and strengths of various variable combinations.

6 Discussion

This chapter summarizes the findings, compares them with the theory, determines the result of the hypothesis testing, and discusses contributions, future research, and limitations.

6.1 Findings

The result is treated as a meaningful finding if it has a stronger than moderate correlation. A self-defined threshold for correlation strength is used, as defined in chapter 5. The findings, as presented in table 1, suggest that there are multiple meaningful findings with moderate and strong correlations.

- Popularity and Activity correlate strongly. When a software project is more active, it is generally also more popular.
- Popularity and Self-Written LOC are moderately correlated. Software projects that have more self-written code are generally also more popular.
- Activity and Self-Written LOC correlating strongly. Active software projects have more self-written code.
- Maturity and Library LOC are moderately correlated. Mature projects also have more third-party code.
- Activity and Library LOC have a moderate correlation. Active projects have more third-party code.

6.1.1 Summary and Hypothesis

Findings suggest that active projects have more library and self-written code within their codebases, so active projects are generally larger. Active projects also correlate highly with popularity, but noteworthy is that popularity does not correlate strongly with the amount of third-party code. Popular projects are generally larger or larger projects that are actively developed and are popular. The proportion of third-party or self-written code does not correlate with the activity or popularity but weakly correlates with maturity. The correlation between maturity and code proportions must be more significant to be deemed a meaningful finding. Mature projects correlate moderately with quantity but not with the proportion of third-party code.

Proportions do not generally correlate with the studied variables, but the quantity of code amount is. This indicates that the proportion of lines of code is pretty stable when the project grows in popularity and activity. In other words, a stable proportion of self-written and library code is kept when projects are active and grow in size and popularity.

Reflecting on the research question and hypotheses, technically, we do have more evidence that the amount of third-party code affects the quality of the software project. However, the finding is not very practical. Results do not answer whether using third-party code positively or negatively contributes to the quality. When the quantity of the third-party code seems to correlate with the quality, the proportion of the third-party code does not. Results implicate that projects with a lot of self-written code are not systematically higher quality than projects with a lot of third-party code.

Findings implicate that high-quality codebases have various proportions of third-party and self-written code, and the proportion is not a decisive factor that affects quality. Results suggest that larger, more popular, and active projects are generally high quality, which is also very intuitive. However, we might lack a variable that expresses a causative relationship with the quality. Even if larger projects are generally higher quality, does the project become high quality if, for example, one person creates a large enough project?

Every relationship reported here is statistically more significant than the chosen α of 0.05, which means that the results did not occur by chance. There is some correlation, but it cannot be clearly stated that third-party code is a factor in the quality of a software project. Therefore H_0, *"Utilizing third-party libraries in software projects developed in 'Go' does not affect the software project's quality."* is accepted.

6.1.2 Contributions and Future Research

The findings of this study challenge the argument from the theory that software reuse is primarily a positive contributor to the quality of the software project. Findings suggest, at least when observing projects written in "Go," that quality is not correlating directly with third-party code proportion. Quality is correlated with other variables, such as the size of the project, popularity, and activity.

Future research could seek causative relationships with quality. This study can be used as a starting point, contributing knowledge of which variables correlate with third-party code in software projects written in Go. Composite variables can also be enhanced to better capture the quality of the software project.

Research could also be conducted on different ecosystems in comparison. Software reuse practices might be completely different in C-family languages that have been around for decades or in the JavaScript ecosystem, known for the notorious amount of libraries and frameworks.

6.2 Limitations

This section discusses limitations that may have affected the study. The limitations may be related to, for example, the interpretation of results, the execution of test methods or the implementation of research software.

Researcher Bias.

Software engineer background and previous experience with the Go programming language might affect how the research was conducted. The researcher might already have an opinion on a topic, which could involve, for example, the decision with the hypothesis.

Standard Library and Other Libraries.

The study did not differentiate the standard library from other libraries. There could be a difference in terms of quality within these, which might affect the results.

Sample and Go Ecosystem.

As a young programming language, the sample is affected by the factor that libraries might not be as mature as in other languages that have been around for a while.

Causative Relationships.

While the study researches correlations between variables, conclusions should be drawn with caution because correlation always does not imply causation between variables. A third variable could have a causative relationship with the variables, and the correlation is just an outcome.

Quality Indicators.

Quality indicators or composite variables are self-defined and composed of multiple variables. Variables might not capture the quality of the software project.

Research Tools.

Self-developed research tools and data sources, Sourcegraph API, might contain bugs that might affect the accuracy and constancy of the results and dataset.

7 Conclusions

This chapter discusses the conclusions of the study. The objective was to study whether opportunistic design patterns are visible from software developed in Go. The research was conducted as a quantitative study using statistical research methods such as hypothesis testing. Two different research tools were developed alongside the study, which are publicly visible on GitHub.

The sample that was studied consists of 2000 open-source software repositories developed in "Go" from GitHub with at least 100 stargazers. Metadata from these repositories was used to capture desired attributes of software projects within multiple composite variables. The desired qualities of the software project were used to form composite variables "Maturity," "Popularity," and "Activity." Desired qualities, as presented in the chapter 3:

- Project has decent documentation available.
- Project passes manual code review.
- Project has an acceptable support level.
- Project is actively developed.
- Project is used by others, or in other words, is popular.
- Project has not a trend with vulnerabilities.
- Project has a license that allows software reuse.
- Project has an acceptable level of transitive dependencies.

The theory argues that systematic software reuse might positively impact the software quality. The theory is in conflict with the findings of this study, at least partly. According to the results, there seems not to be a correlation between the proportion of reused code (in the form of libraries) and the defined quality variables of the software when studying software developed in Go. Opportunistic design patterns cannot be identified from the studied projects.

While proportion was not the decisive factor, findings suggest a clear correlation between the size of the repository, popularity, and activity. In other words, popular and more actively followed and developed projects are often more high-quality, at least in the case of this sample. However, when reflecting on these findings, the sheer quantity of code and its correlation with quality does not feel intuitive. There is a probability that this relationship is not causative. For example, if the author creates a huge codebase, what guarantees that the codebase is then automatically high quality? Checking causation or searching for variables that have a causal relationship with quality could be a future research topic.

Limitations that might affect this study are mainly human errors. Also, one of the big factors might be that there was no differentiation between a standard library and other third-party libraries.

Bibliography

Anderson, David R., Dennis J. Sweeney, and Thomas A. Williams. 2023. "Hypothesis testing". Visited on February 19, 2023. https://www.britannica.com/science/statistics/Residual-analysis.

Aris Papadopoulo. 2020. *Should Developers Use Third-Party Libraries?* Visited on February 18, 2023. https://www.scalablepath.com/back-end/third-party-libraries.

Asq. 2023. "Learn About Quality: What is Software Quality?" Visited on May 29, 2023. https://asq.org/quality-resources/software-quality.

Bevans, Rebecca. 2022. "Choosing the Right Statistical Test: Types and Examples", visited on February 20, 2023. https://www.scribbr.com/statistics/statistical-tests/.

Bhandari, Pritha. 2021. "Correlation Coefficient: Types, Formulas and Examples", visited on April 30, 2023. https://www.scribbr.com/statistics/correlation-coefficient/.

———. 2022a. "Correlational Research: Guide, Design and Examples", visited on May 5, 2023. https://www.scribbr.co.uk/research-methods/correlational-research-design/.

———. 2022b. "What Is Quantitative Research? Definition, Uses and Methods", visited on April 27, 2022. https://www.scribbr.com/methodology/quantitative-research/.

Carter, John. 2023. "Time To Market (TTM) Defined and Why It Is Important". Visited on August 13, 2023. https://www.tcgen.com/time-to-market/.

Cemazar, Sara Ana. 2022. "10 biggest advantages of open-source software". Visited on March 2, 2023. https://de.rocket.chat/blog/open-source-software-advantages.

Ciaburro, Giuseppe, V. Kishore Ayyadevara, and Alexis Perrier. 2018. *Hands-On Machine Learning on Google Cloud Platform.* Chapter: Min–max normalization. Packt Publishing. ISBN: 1788393481.

Cornell University. 2010. "Lecture 18: Concurrency - Producer / Consumer Pattern and Thread Pools". Visited on February 8, 2023. https://www.cs.cornell.edu/courses/cs3110/2010fa/lectures/lec18.html.

Cox, Russ. 2019. "Surviving Software Dependencies". *Association for Computing Machinery* 17 (2): 24–47.

Dan’s Tools. 2023. “The Current Epoch Unix Timestamp”. Visited on August 28, 2023. https://www.unixtimestamp.com/.

edvantis. 2020. “Quality Assurance vs Quality Control: Key Differences Explained”. Visited on August 22, 2023. https://www.edvantis.com/blog/qa-vs-qc/.

freeCodeCamp. 2023. “What is ORM? The Meaning of Object Relational Mapping Database Tools”. Visited on September 7, 2023. https://www.freecodecamp.org/news/what-is-an-orm-the-meaning-of-object-relational-mapping-database-tools/.

Galin, Daniel. 2018a. *Software Quality: Concepts and Practice.* Chapter 1.1 Software Quality and Software Quality Assurance –Definitions. Wiley IEEE. ISBN: 978-1-119-13449-7.

———. 2018b. *Software Quality: Concepts and Practice.* Chapter 1.4 Software Errors, Faults, and Failures. Wiley IEEE. ISBN: 978-1-119-13449-7.

———. 2018c. *Software Quality: Concepts and Practice.* Chapter 1.6 Software Quality Assurance Versus Software Quality Control. Wiley IEEE. ISBN: 978-1-119-13449-7.

———. 2018d. *Software Quality: Concepts and Practice.* Chapter 1.7 Software Quality Engineering and Software Engineering. Wiley IEEE. ISBN: 978-1-119-13449-7.

———. 2018e. *Software Quality: Concepts and Practice.* Chapter 2.2 The Need for Comprehensive Software Quality Requirements. Wiley IEEE. ISBN: 978-1-119-13449-7.

———. 2018f. *Software Quality: Concepts and Practice.* Chapter 2.3 Mc Call’s Classic Model for Software Quality Factors. Wiley IEEE. ISBN: 978-1-119-13449-7.

———. 2018g. *Software Quality: Concepts and Practice.* Chapter 2.3.1 Mc Call’s Product Operation Software Quality Factors. Wiley IEEE. ISBN: 978-1-119-13449-7.

———. 2018h. *Software Quality: Concepts and Practice.* Chapter 2.4.1 The ISO/IEC 25010 Model. Wiley IEEE. ISBN: 978-1-119-13449-7.

———. 2018i. *Software Quality: Concepts and Practice.* Chapter 2.4.2 Alternative Software Quality Models. Wiley IEEE. ISBN: 978-1-119-13449-7.

GeeksForGeeks. 2023. “Lines of Code (LOC) in Software Engineering”. Visited on August 13, 2023. https://www.geeksforgeeks.org/lines-of-code-loc-in-software-engineering/.

George, Tegan, and Shona McCombes. 2022. "How to Write a Thesis or Dissertation Conclusion", visited on May 28, 2023. https://www.scribbr.com/dissertation/write-conclusion/.

GitHub. 2023a. "GitHub". Visited on August 28, 2023. https://github.com.

———. 2023b. "GitHub Docs: About issues". Visited on September 5, 2023. https://docs.github.com/en/issues/tracking-your-work-with-issues/about-issues.

———. 2023c. "GitHub Docs: About commits". Visited on September 6, 2023. https://docs.github.com/en/pull-requests/committing-changes-to-your-project/creating-and-editing-commits/about-commits.

———. 2023d. "GitHub Docs: Fork a repo". Visited on September 6, 2023. https://docs.github.com/en/get-started/quickstart/fork-a-repo.

———. 2023e. "GitHub Docs: About pull requests". Visited on August 28, 2023. https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests.

———. 2023f. "GitHub Docs: About releases". Visited on September 6, 2023. https://docs.github.com/en/repositories/releasing-projects-on-github/about-releases.

———. 2023g. "GitHub Docs: Events". Visited on September 6, 2023. https://docs.github.com/en/rest/activity/events?apiVersion=2022-11-28.

———. 2023h. "GitHub Docs: Managing your subscriptions". Visited on September 6, 2023. https://docs.github.com/en/account-and-profile/managing-subscriptions-and-notifications-on-github/managing-subscriptions-for-activity-on-github/managing-your-subscriptions.

———. 2023i. "GitHub Docs: Contributing to projects". Visited on September 6, 2023. https://docs.github.com/en/get-started/quickstart/contributing-to-projects.

———. 2023j. "haapjari/draw". Visited on February 9, 2023. https://github.com/haapjari/draw.

———. 2023k. "haapjari/glass". Visited on February 6, 2023. https://github.com/haapjari/glass.

———. 2023l. "hhatto/gocloc". Visited on February 6, 2023. https://github.com/hhatto/gocloc.

Go. 2023. "A Tour of Go: Goroutines". Visited on February 8, 2023. https://go.dev/tour/concurrency/1.

GraphQL. 2023. "GraphQL: A query language for your API". Visited on February 14, 2023. https://graphql.org/.

Gravetter, Frederick J., and Larry B. Wallnau. 2017. *Statistics for the Behavioral Sciences.* 225. Cengage Learning.

Hartmann, Björn, Scott Doorley, and Scott R. Klemmer. 2008. “Hacking, Mashing, Gluing: Understanding Opportunistic Design”. *IEEE Pervasive Computing,* 1–9.

Henkel, Ramon. E. 1976a. *Tests of Significance.* 44. Beverly Hills: Sage Publications, Inc. ISBN: 0803906528. https://doi.org/10.4135/9781412986113.

———. 1976b. *Tests of Significance.* 35–42. Beverly Hills: Sage Publications, Inc. ISBN: 0803906528. https://doi.org/10.4135/9781412986113.

Krueger, Charles W. 1992. “Software Reuse”. *ACM Computing Surveys (CSUR)* 24 (2): 179.

LawInsider. 2023. “Software Module definition”. Visited on August 21, 2023. https://www.lawinsider.com/dictionary/software-module.

Madhugiri, Devashree. 2023. “Exploratory Data Analysis: Types, Tools, Process”. Visited on May 18, 2023. https://www.knowledgehut.com/blog/data-science/eda-data-science.

Magiya, Joseph. 2023. “Kendall Rank Correlation Explained”. Visited on May 29, 2023. https://towardsdatascience.com/kendall-rank-correlation-explained-dee01d99c535.

Mäkitalo, Niko, Antero Taivalsaari, Arto Kiviluoto, Tommi Mikkonen, and Rafael Capilla. 2020. “On opportunistic software reuse”. *Computing* 102 (11): 2385–2386.

Martin, James. 1983. *Managing the Data-base Environment.* 381. Englewood Cliffs, N.J : Prentice-Hall. ISBN: 0-135-50582-8. https://archive.org/details/managingdatabase00mart/page/381/mode/2up.

McCombes, Shona. 2022. “How to Write a Discussion Section: Tips and Examples”, visited on May 28, 2023. https://www.scribbr.com/dissertation/discussion/.

———. 2023. “How to Write a Literature Review: Guide, Examples and Templates”, visited on May 28, 2023. https://www.scribbr.com/methodology/literature-review/.

Md. Fahim Bin Amin. 2023. “How to Add Stargazers and Forkers Cards to Your GitHub Repository”. Visited on September 6, 2023. https://www.freecodecamp.org/news/how-to-add-stargzers-and-forkers-to-your-github-repository/.

Metrics Toolkit. 2023. “GitHub: Forks, collaborators, wachers”. Visited on October 8, 2023. https://www.metrics-toolkit.org/metrics/github_forks_collaborators_watchers/.

Mikkonen, Tommi, and Antero Taivalsaari. 2019. “Hacking, Mashing, Gluing”. *IEEE Software* 36 (3): 105–111.

MOOC.fi. 2023. “MOOC.fi: Data Analysis with Python”. Visited on September 7, 2023. https://courses.mooc.fi/org/uh-cs/courses/dap-22.

OpenSource. 2023. “What is Open Source?” Visited on August 13, 2023. https://opensource.com/resources/what-open-source.

Pandey, Sakshi. 2022. “The Importance Of Code Reusability In Software Development”. Visited on September 20, 2023. https://www.browserstack.com/guide/importance-of-code-reusability.

PennState Eberly College of Science. 2023. “Statistics Online: S.3.2. - Hypothesis Testing (P-Value Approach)”. Visited on February 20, 2023. https://online.stat.psu.edu/statprogram/reviews/statistical-concepts/hypothesis-testing/p-value-approach.

ProductPlan. 2023. “Technical Debt: What is Technical Debt?” Visited on August 22, 2023. https://www.productplan.com/glossary/technical-debt/.

Pronschinske, Mitch. 2016. “A General Software Maturity Model: Learn about the benefits of a mature project, the pitfalls, and how to qualitatively and quantitatively analyze a project’s maturity level in the correct context.” Visited on March 2, 2023. https://dzone.com/articles/a-general-software-maturity-model.

Python. 2023. “Python”. Visited on September 7, 2023. https://www.python.org/.

RedHat. 2023. “What is an API?” Visited on August 13, 2023. https://www.redhat.com/en/topics/api/what-are-application-programming-interfaces.

Rouse, Margaret. 2016. “Software Library: What Does Software Library Mean?” Visited on August 16, 2023. https://www.techopedia.com/definition/3828/software-library.

———. 2022. “Software Package: What Does Software Package Mean?” Visited on August 21, 2023. https://www.techopedia.com/definition/4360/software-package.

Saied, Mohamed Aymen, Ali Ouni, Houari Sahraoui, Raula Gaikovina Kula, Katsuro Inoue, and David Lo. 2018. “Improving reusability of software libraries through usage pattern mining”. *Journal of Systems and Software,* 1.

Sharot, Tali. November 2004. “Mindless Statistics”. *The Journal of Socio-Economics* 33 (5): 587–606. https://doi.org/https://doi.org/10.1016/j.socec.2004.09.033.

Sommerville, Ian. 2015. *Software Engineering.* Chapter 15 Software Reuse. Pearson. ISBN: 9332582696.

SonaType. 2023. “What Are Software Dependencies?” Visited on August 16, 2023. https://www.sonatype.com/launchpad/what-are-software-dependencies.

Sourcegraph. 2023. “Sourcegraph docs: Sourcegraph GraphQL API”. Visited on February 5, 2023. https://docs.sourcegraph.com/api/graphql.

StackOverflow. 2023. “2022 Developer Survey”. Visited on May 9, 2023. https://survey.stackoverflow.co/2022/#overview.

Statology. 2019. “Pearson Correlation Coefficient”. Visited on May 7, 2023. https://www.statology.org/pearson-correlation-coefficient/.

Tietoarkisto. 2023. “Hypoteesien testaus: Tilastollisten testien kritiikki”. Visited on February 13, 2023. https://www.fsd.tuni.fi/fi/palvelut/menetelmaopetus/kvanti/hypoteesi/testaus/#kritiikki.

Turney, Shaun. 2022. “Pearson Correlation Coefficient: Guide and Examples”, visited on May 7, 2023. https://www.scribbr.com/statistics/pearson-correlation-coefficient/.

Vial, Gregory. 2022. “Manage the Risks of Software Reuse”. *MIT Sloan Management Review,* visited on June 20, 2023.