Analyzing Program Comprehensibility of Go Projects

Moumita Asad, Rafed Muhammad Yasir, Shihab Shahriar, Nadia Nahar, Md. Nurul Ahad Tawhid
Institute of Information Technology, University of Dhaka, Bangladesh
{*bsse0731, bsse0733, bsse0703, nadia, tawhid*}*@iit.du.ac.bd*

***Abstract*—Program comprehension is one of the most important activities in developing and maintaining software. Although existing studies have examined aspects of Go such as design patterns, code smells and comment density, the comprehensibility of Go has not been explored yet. This study analyzes the comprehensibility of Go by comparing it with Java based on five metrics namely Too Long Files, Too Long Methods, Nesting Depth, Lack of Cohesive Comments and Duplicate Comments. For comparison, 50 popular, diverse and open-source projects are selected from each language. Results show that Go projects outperform Java in terms of Nesting Depth, Lack of Cohesive Comments and Duplicate Comments.**

***Index Terms*—Go, Java, Program comprehensibility, Software maintenance.**

I. Introduction

Go is an open-source programming language developed and maintained by Google [1]. At present, the popularity of Go is rising rapidly and many teams are adopting it for their projects. Renowned projects such as Docker and Kubernetes have been written in Go. Existing studies [1], [2], [3] have analyzed aspects of Go such as design patterns, code smells and comment density. However, the comprehensibility (the degree of ease with which a programmer read and comprehend a program [4]) of Go projects have not been explored yet.

Program comprehension is the precondition of performing any maintenance related activities [5]. It consumes more than half of the maintenance resources [6]. A major reason behind introducing the Go programming language is to make code more maintainable [7]. To make Go projects maintainable, code must be comprehensible [4]. Therefore, analyzing the program comprehensibility of Go projects will help to gain insight regarding their maintainability and identify scopes for making the projects more comprehensible. Furthermore, it will help developers to decide whether they should use Go for their projects [6].

In this context, the current study aims at investigating the comprehensibility of projects written in Go. Additionally, a tool named *CompreGo* has been developed to detect code fragments that need to be refactored for improving comprehensibility. To analyze the comprehensibility of Go, it is compared with Java. The reason is that both of these languages are object-oriented languages [2] and both of these belong to the C-family (e.g., C++, C#) [8]. Furthermore, an existing study concluded that Java projects have higher program comprehensibility compared to C, C++ and C# projects [9]. This study measures program comprehensibility using five static code metrics namely Too Long Files, Too Long Methods, Nesting Depth, Lack of Cohesive Comments (non-informative comments) and Duplicate Comments [9], [10]. These metrics are language-independent and found to be a good indicator of program comprehensibility. Based on these metrics, Wilcoxon Rank-Sum Test [11] is used to analyze comprehensibility of Go and Java projects.

For analysis, 50 popular, diverse and open-source projects are selected from each language based on several criteria such as projects cannot be a fork, and it must have a description or readme file [9]. Results show that Go projects have significantly better comprehensibility than Java in terms of Nesting Depth, Lack of Cohesive Comments and Duplicate Comments. However, Java projects have significantly fewer long methods than Go. Regarding Too Long Files, no significant difference is found between the two languages.

II. Methodology

This study analyzes the program comprehensibility of Go projects by comparing those with Java projects. The study is conducted in 4 steps namely dataset selection, dataset preprocessing, comprehensibility metrics calculation and statistical analysis. At first, Go and Java projects are selected based on several criteria such as availability of readme files. Next, relevant files (e.g., production code) are selected from the projects to achieve better comparability of results [9]. After that, comprehensibility metrics of the code residing in these files are calculated. Based on these metrics, statistical analysis is performed to identify if the comprehensibility of Go and Java projects are significantly different. The details of these steps are given below:

1) **Dataset Selection:** This study selects Go and Java projects that fulfill the following criteria:
   - To filter out dummy projects, repositories having at least 10,000 lines of code and a readme file or a GitHub description are selected, as followed in [9].
   - To avoid duplicate projects, repositories must not be a fork or mirror of another one.
   - An existing study found that almost 75% projects above 215 MB contain the same code multiple times [9]. Besides, 99% repositories in the GHTorrent dataset (a collection of GitHub repositories) [12]

DOI reference number: 10.18293/SEKE2021-152

are below this threshold. Therefore, the size of the codebase should not exceed this threshold.

2) **Dataset Preprocessing:** After selecting the projects, folders containing external libraries, generated source files or test code are excluded to solely focus on the core code of the repository. Although test codes are important, those are excluded from analysis to improve comparability between the repositories because it is found that developers may not apply high quality standards to it [9]. To filter out these files, a list of 60 file system paths (e.g., "**/libs/**", "**/test/**", "**/testsuite/**") provided in [9] is used. The entities of the list do not depend on the organization or naming convention followed by the Go or Java repositories. Apart from the list, the *pkg* folder is excluded from Go projects since it contains internal libraries that are not part of the main source code.
3) **Metrics Calculation:** To measure program comprehensibility, five static code metrics namely Too Long Files, Too Long Methods, Nesting Depth, Lack of Cohesive Comments and Duplicate Comments are used. These metrics are language-independent and found to be a good indicator of program comprehensibility [9]. These metrics are calculated by inspecting source code files and parsing the corresponding Abstract Syntax Tree.

- **Too Long Files (TLF):** Long files are difficult to comprehend since a large fragment of code needs to be reviewed [9]. These files can indicate bad modularization as well. Too Long Files measures the portion of source code lines that are located in files exceeding 750 lines [9]. For calculating this metric, (1) is used.

$$TLF = \frac{\sum LOC\ of\ long\ files}{total\ LOC} * 100 \quad (1)$$

Where, $total\ LOC$ denotes LOC of the whole project.

- **Too Long Methods (TLM):** Similar to long files, long methods are difficult to comprehend since a large code fragment needs to be examined [6], [9]. Too Long Methods calculates the number of code lines that reside in methods surpassing 75 lines [9]. It is measured using (2).

$$TLM = \frac{\sum LOC\ of\ long\ methods}{total\ LOC} * 100 \quad (2)$$

- **Nesting Depth (ND):** Deeply nested codes are hard to comprehend as the interleaving control structure of the code needs to be considered, as shown in Listing 1 [13]. Nesting Depth measures the fraction of code lines belonging to methods exceeding nesting depth 5 [9]. It is calculated using (3).

$$ND = \frac{\sum LOC\ of\ deeply\ nested\ methods}{total\ LOC} * 100 \quad (3)$$

```
if(marks<40) {
   result = "fail";
}
else {
   if(marks>=80) {
      result = "A";
   }
   else if(marks>=60) {
      result = "B";
   }
   else {
      result = "C";
   }
}
```

Listing 1. Nesting Depth

- **Lack of Cohesive Comments (LCC):** Comments lack cohesiveness when they are non-informative [10]. A sample comment is shown in Listing 2, where the comment conveys obvious information. Lack of cohesive comments can confuse and mislead developers and increase the cost to comprehend and maintain code [14].

```
/**
* @return true or false
*/
protected boolean
    isLoginTicketBased() throws
    Exception {
   ..........
   ..........
}
```

Listing 2. Non-cohesive Comment

To compute Lack of Cohesive Comments, the name and lead comment of each method are tokenized using camel case format and white-space respectively [14]. Next, the similarity between these two token sets is measured using Levenshtein distance [15]. Two tokens are considered similar if the distance is smaller than 2 [10]. To calculate coherence, the total number of similar words are divided by the total number of comment words. If the value of coherence is 0 or above 0.5, the comment is non-cohesive [10]. Lastly, Lack of Cohesive Comments is calculated using (4).

$$LCC = \frac{total\ non\text{-}cohesive\ comments}{total\ comments} * 100 \quad (4)$$

Where, $total\ comments$ represents the total number of method lead comments in the project.

- **Duplicate Comments (DC):** Existing studies show that source code often contains methods with duplicate comments [14], [16]. These comments do not provide meaningful information on the different implementation of methods and thereby increases the difficulty in program comprehension [14]. Calculation of Duplicate Comments follows a similar process as Lack of Cohesive Comments. Duplicate

comments are searched within a file since they are more likely to occur in the same file [16]. At first, all the comments of a file are tokenized and stored in lists. For each comment pair, Levenshtein distance [15] is calculated between elements of their corresponding token lists. Two comments are considered duplicate if the distances of all the elements are less than 2 [10]. Lastly, Duplicate Comments are measured using (5).

$$DC = \frac{total\ duplicate\ comments}{total\ comments} * 100 \quad (5)$$

All of the above metrics are measured in percentage and a higher value indicates lower comprehensibility.

4) **Statistical Analysis**: After calculating the metrics, Wilcoxon Rank-Sum Test [11] is conducted to check whether the comprehensibility of Go and Java projects are significantly different. Wilcoxon Rank-Sum Test is used because it makes no assumption regarding the distribution of the data (e.g., normal distribution) and it can handle unequal sample size (e.g., different number of duplicate comments for Go and Java projects) [11]. The null hypothesis is "Code written in Go and Java have no differences regarding comprehensibility". This hypothesis is tested individually for each of the five comprehensibility metrics discussed above.

III. Supporting Tool: CompreGo

To calculate comprehensibility metrics for Go, a tool named *CompreGo* is developed which is publicly available [1]. Developers can use the tool to measure the comprehensibility of a project and identify which fragments need to be refactored for improving comprehensibility.

- **Comprehensibility Metrics of a Project:** To find comprehensibility metrics (Too Long Files, Too Long Methods, Nesting Depth, Lack of Cohesive Comments and Duplicate Comments) of a project, run the following command, where *directory* is the project path.

```
./comprego -d directory
```

The output will show metrics value in percentage, as displayed in Fig. 1. Here, Too Long Files, Too Long Methods, Nesting Depth, Lack of Cohesive Comments and Duplicate Comments are 15.92%, 10.69%, 4.55%, 17.45% and 6.06% respectively.

```
--- Project Comprehensibility Metrics ---
Too Long Files           : 15.92
Too Long Methods         : 10.69
Nesting Depth            : 04.55
Lack of cohesive comments: 17.45
Duplicate comments       : 06.06
```

Fig. 1. Comprehensibility Metrics of a Project

[1]https://github.com/rafed/CompreGo

A number of optional arguments can be specified with this command.

- "*-TLF*": View list of too long files
- "*-TLM*": View list of too long methods
- "*-ND*": View list of deeply nested methods
- "*-LCC*": View list of non-cohesive comments
- "*-DC*": View list of duplicate comments
- "*-ALL*": View details of all the five metrics

A sample output for "*-TLM*"command is shown in Fig. 2. Here, *cleanPath()* method from file *gin-1/4.0/path.go* is a long method since its LOC is 90.

```
Too Long Methods:
  gin-1.4.0/path.go:cleanPath()  len:90
  gin-1.4.0/tree.go:addRoute()  len:128
  gin-1.4.0/tree.go:insertChild()  len:112
  gin-1.4.0/tree.go:getValue()  len:136
  gin-1.4.0/tree.go:findCaseInsensitivePath()  len:115
```

Fig. 2. List of Long Methods

- **Custom Thresholds for Metrics Calculation:** The default thresholds for Too Long Files, Too Long Methods and Nesting Depth are 750, 75 and 5 respectively [9]. However, each project and its maintainers are different and thus thresholds may need to be changed. For this reason, *CompreGo* has a provision for customizing these thresholds. These thresholds can be changed by adding the following arguments while running the tool:
  - "*-lf number*": Set customized threshold for Too Long Files
  - "*-lm number*": Set customized threshold for Too Long Methods
  - "*-nd number*": Set customized threshold for Nesting Depth

  Where, *number* represents the customized threshold value.

IV. Experimentation and Result Analysis

For comparing the comprehensibility of Go and Java, 50 popular, open-source projects of each language are selected from GitHub that fulfill the criteria presented in Section 2 [17]. The popularity of a project is measured based on the number of stars, as followed in [18], [19]. Table I shows the top 5 projects from each language along with their number of stars and LOCs. For example, project *Guava* has 40000 stars and its LOC is 756954.

To compare the projects, Wilcoxon Rank-Sum Test [11] is conducted separately for each of the five comprehensibility metrics (Too Long Files, Too Long Methods, Nesting Depth, Lack of Cohesive Comments and Duplicate Comments), discussed in Section 2. The results are presented in Table 2.

Although the mean of Too Long Files is slightly higher in Go, there is no significant difference between the two languages (p-value = 0.39 $>$ 0.05). In terms of Too Long Methods, Go has longer methods than Java. This is because

TABLE I
DESCRIPTION OF TOP 5 JAVA AND GO PROJECTS

Language	Project	Star	LOC
Java	Elasticsearch	53200	2566030
	Spring Boot	53000	563943
	RxJava	44100	467397
	Guava	40000	756954
	Retrofit	37400	36610
Go	Moby	59400	1396099
	Hugo	49400	144222
	Gin	44800	15968
	Fro	42300	20771
	Gogs	36200	90034

of the error handling mechanism followed by these two languages. In Java, errors are mostly dealt with *try catch* blocks. On the contrary, errors in Go are handled using *if* statements. Whenever a function is called in Go, it usually returns an error in its return values [8]. The errors returned by the callee function are handled with an *if* block inside the caller function. For example, the function *PreparedQueryResolve()* in Listing 3 returns *ErrMissingQueryID* as an error and the caller function *Explain()* in Listing 4 handles the error using an *if* statement. Although Java can handle multiple errors using one *catch* block, Go associates individual *if* statement for each function call. Consequently, more error handling statements are written inside Go method than Java which increases the method size. Through analysis, it is found that on average each Java method contains 0.05 error handling statements (*catch* block), whereas it is 0.88 in Go. Besides, Go does not support generics for increasing simplicity and making code more readable. However, this also makes the code more verbose compared to other languages and results in longer files and methods [2].

TABLE II
RESULT OF WILCOXON RANK-SUM TEST

Metric	Mean (Go)	Mean (Java)	P-value
Too Long Files	25.61	18.99	0.39
Too Long Method	17.59	5.69	0.00
Nesting Depth	3.78	4.49	0.00
Lack of Cohesive Comments	18.47	25.69	0.00
Duplicate Comments	2.89	5.43	0.00

[1] The significance level is 0.05

In terms of Nesting Depth, Go code have fewer nesting than Java. The reason again lies in the way how errors are handled in these languages. In Java, *try* blocks are used to handle code that may throw an error. When a *try* block is added in a method, it increases the nesting depth, as shown in Listing 5. Results reveal that around 33.14% deeply nested method in Java contains at least one *try catch* block. To find the association between deeply nested method and the presence of *try catch* block, Chi-square Test [20] is conducted since both of these are categorical variables. The result shows a significant association between deeply nested method and the presence of *try catch* block (p-value < 0.05). However, there is no concept of *try catch* blocks in Go [8]. Rather, errors are handled by adding *if* statement inside the caller function. The *if* statement checks the error and return values accordingly, as presented in Listing 4. Since the *if* statement exist in the same nesting depth as it's neighbouring code, it does not increase the nesting depth of the function. As a result, functions in Go are less nested than Java.

```
func (s *Store) PreparedQueryResolve()
    (uint64,*structs.PreparedQuery, error) s{
  if queryIDOrName == "" {
    return 0, nil, ErrMissingQueryID
  }
}
```

Listing 3. Returning error in Go

```
func (p *PreparedQuery) Explain() error {
   _, query, err :=
       state.PreparedQueryResolve()
   if err != nil {
      return err
   }
}
```

Listing 4. Handling error using *if* statement in Go

```
@Override public void run(){
   while (!__isClosed) {
      try {
         if ((ch=__read(true)) < 0) {
            break;
         }
      }
      catch (InterruptedIOException e) {
         synchronized (__queue) {
            __ioException=e;
            __queue.notifyAll();
         }
      }
   }
}
```

Listing 5. Handing error using *try catch* blocks in Java

Go shows better results in Lack of Cohesive Comments and Duplicate Comments as well. This can be due to the difference in documentation generation process from the source code comments [21]. Java and Go use different comment structures and separate tools to generate documentation. In Java, documentation is generated through the Javadoc tool [21]. It is necessary for comments in Java to be annotated with Javadoc tags (e.g., @param, @return, @author) to produce meaningful documentation. In Go, documentation is generated through the Godoc tool [8]. Unlike Javadoc, Godoc does not require tags in comments for generating meaningful documentation. It only needs a comment preceding a code and its description in plain words. Godoc then uses the comment and as much information it can get from the code to generate documentation. This makes the process of documenting Go code much easier than Java. Additionally, the community considers that the process of generating documentation from comments is simpler and

easier in Go than in other languages[2]. Such advantages may motivate developers to write better comments that are more cohesive and not duplicates of other comments.

V. Threats to Validity

This section presents potential aspects that may threaten the validity of the study:

- **Threats to external validity:** The analysis is conducted on 50 Go and 50 Java projects which are selected using several criteria such as availability of readme file or having at least 10,000 lines of code. Although the project selection criteria are based on an existing study [9], the obtained results may not generalize to other projects. However, to mitigate the threat of generalizability, diverse and popular projects are selected, as followed in [17], [22].
- **Threats to internal validity:** Threats to internal validity include errors in the implementation and experimentation. The first threat to internal validity lies in measuring program comprehension. Although research on program comprehension started more than 30 years ago, till now there is no well-defined metric to measure it [23]. This study uses five static code metrics namely Too Long Files, Too Long Methods, Nesting Depth, Lack of Cohesive Comments and Duplicate Comments to measure program comprehensibility. An existing study found that these metrics are language-independent and a good indicator of program comprehensibility [9]. However, changing the metrics (e.g., psychological factors, dynamic metrics [23]) may impact the obtained results. The second threat to internal validity comes from setting the thresholds of the comprehensibility metrics. This study set the threshold for Too Long Files, Too Long Methods and Nesting Depth to 750, 75 and 5 respectively. These thresholds may vary depending on the context. However, these thresholds are adopted by a previous study [9] as well.

VI. Related Work

Existing work related to this paper can be broadly classified into two categories - studies related to program comprehension and studies related to Go, which are discussed in the following subsections.

A. Studies Related to Program Comprehension

Although research on program comprehension started more than 30 years ago, there are still no well-defined metrics to measure it [23]. However, various researches [5], [9], [24] have been conducted in this regard. Scalabrino et al. used 121 metrics to investigate their correlation with program comprehension [5]. These metrics are related to code (e.g., cyclomatic complexity), documentation (e.g., methods internal documentation quality) and developer experience (e.g., years of programming experience in any language). However, the study found that none of the metrics show a significant correlation with program comprehension.

[2]https://blog.golang.org/godoc-documenting-go-code

Trockman et al. reanalyzed the data from the study of Scalabrino et al. [5] using principal component analysis and stepwise selection [24]. They did not find a specific principal component that explains most of the variance. Stepwise selection revealed that an increase in methods internal documentation quality, max. line length and larger number of periods per line correlate with lower understandability. However, the authors pointed out that the dataset is not large enough (only 324 observations from 46 developers) to draw any conclusion.

Roehm et al. studied 10 conventional wisdom related to software maintainability more specifically program comprehension [9]. For example, C code has more too long methods than code written in other languages. They used 5 metrics - Clone Coverage, Comment Incompleteness, Too Long Files, Too Long Methods and Nesting Depth for measuring program comprehension. However, another study countered that code clones are helpful for program comprehension [25]. In addition, writing comments are not enough if they convey unrelated or inconsistent information [14]. Therefore, this paper uses Lack of Cohesive Comment and Duplicate Comments instead of Comment Incompleteness. Furthermore, unlike this study, their analysis does not consider Go programming language.

B. Studies Related to Go

Prior studies related to Go can be divided into two categories. The first category [3], [22] includes Go projects as a part of their dataset. Ray et al. compared 17 programming languages including Go to find whether the choice of language affects software quality [22]. Their analysis confirms that language has a small yet statistically significant impact on code quality. Besides, classifying bugs into several categories (e.g., Memory or Concurrency error), the paper examined whether language influences the type of bug that occurs in software. The result revealed that defect types are strongly associated with languages. For example, languages with managed memory systems (e.g., Java) naturally had fewer memory errors or leaks compared to unmanaged languages (e.g., C).

Another study conducted by He et al. explored differences in commenting practices across different languages [3]. They analyzed the comment density of 5 popular programming languages namely Python, Java, Go, JavaScript and C++. Their study revealed that Python and Java projects have significantly higher comment density than C++, JavaScript and Go projects. In addition, the purpose of a project (e.g., reuse, application, education) impacts its comment density. For example, educational projects have the highest comment density.

The second category [1], [2] conducts research on Go from various perspectives. Schmager et al. analyzed the design patterns of Go [2]. They implemented all the 23 Gangs of Four (GoF) design patterns and compared these with Java. They found that Go's language features have not replaced design patterns. Implementing the adapter pattern is easier in Go. On the other hand, implementing the template pattern is difficult since there is no abstract class in Go. Furthermore, although Go syntax is an improvement over C++ or Java, it is more verbose than Python or Haskell.

Another study by Yasir et al. proposed a tool named GodExpo to detect God Structures (a structure that threatens the maintainability and understandability of code by performing most of the work alone) in Go programs [1]. GodExpo uses three metrics namely Weighted Method Count (WMC), Tight Class Cohesion (TCC) and Access To Foreign Data (ATFD) for detecting God Structures. Besides, it can provide version wise result to observe the evolution of God Structures. By executing GodExpo on Go projects, the authors showed that it can detect God Structs in all versions of a project. Additionally, it was found that number of God Structures in a project gradually increases as a project evolves.

The above discussion indicates that various studies have been conducted on Go. However, none of these studies focuses on the maintainability or comprehensibility of Go. Therefore, this paper aims at analyzing the comprehensibility of Go. In addition, it proposes a tool *CompreGo* to help developers in tracking the comprehensibility of Go projects.

VII. Conclusion and Future Work

This paper examines the comprehensibility of Go projects by comparing it with Java projects. For measuring program comprehensibility, five static code metrics namely Too Long Files, Too Long Methods, Nesting Depth, Lack of Cohesive Comments and Duplicate Comments are used. Based on these metrics, 50 popular, diverse, open-source Go and Java projects are compared using Wilcoxon Rank-Sum Test. Results demonstrate that Go code has significantly higher comprehensibility than Java in terms of Nesting Depth, Lack of Cohesive Comments and Duplicate Comments. Conversely, Java has significantly better comprehensibility than Go in terms of Too Long Methods. Regarding Too Long Files, no significant difference is observed between these two languages. In future, the comprehensibility of Go projects will be analyzed from other perspectives such as dynamic analysis or psychological aspect. In addition, refactoring suggestion will be developed for improving comprehensibility of Go projects.

References

[1] Rafed Muhammad Yasir, Moumita Asad, Asadullah Hill Galib, Kishan Kumar Ganguly, and Md Saeed Siddik. Godexpo: an automated god structure detection tool for golang. In *Proceedings of the 3rd International Workshop on Refactoring*, pages 47–50. IEEE Press, 2019.

[2] Frank Schmager, Nicholas Cameron, and James Noble. Gohotdraw: Evaluating the go programming language with design patterns. In *Evaluation and Usability of Programming Languages and Tools*, page 10. ACM, 2010.

[3] Hao He. Understanding source code comments at large-scale. In *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1217–1219. ACM, 2019.

[4] Gerard K Rambally. The influence of color on program readability and comprehensibility. In *Proceedings of the 17th SIGCSE Technical Symposium on Computer Science Education*, pages 173–181, 1986.

[5] Simone Scalabrino, Gabriele Bavota, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk, and Rocco Oliveto. Automatically assessing code understandability: How far are we? In *Proceedings of the 32nd International Conference on Automated Software Engineering*, pages 417–427. IEEE Press, 2017.

[6] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering*, 44(10):951–976, 2017.

[7] Rob Pike. Go at google: Language design in the service of software engineering, 2012. *URL http://talks.golang. org/2012/splash.article*, 2012.

[8] Ivo Balbaert. *The way to Go: A thorough introduction to the Go programming language*. IUniverse, 2012.

[9] Tobias Roehm, Daniel Veihelmann, Stefan Wagner, and Elmar Juergens. Evaluating maintainability prejudices with a large-scale study of open-source projects. In *Proceedings of the International Conference on Software Quality*, pages 151–171. Springer, 2019.

[10] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. In *Proceedings of the 21st International Conference on Program Comprehension*, pages 83–92. IEEE, 2013.

[11] William Cyrus Navidi. *Statistics for engineers and scientists*. McGraw-Hill Higher Education New York, NY, USA, 2008.

[12] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 92–101, 2014.

[13] Jan-Peter Ostberg and Stefan Wagner. On automatically collectable metrics for software maintainability evaluation. In *Proceedings of the Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement*, pages 32–37. IEEE, 2014.

[14] Anna Corazza, Valerio Maggio, and Giuseppe Scanniello. Coherence of comments and method implementations: a dataset and an empirical investigation. *Software Quality Journal*, 26(2):751–777, 2018.

[15] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095, 2007.

[16] Arianna Blasi and Alessandra Gorla. Replicomment: identifying clones in code comments. In *Proceedings of the 26th Conference on Program Comprehension*, pages 320–323. ACM, 2018.

[17] Joao P Diniz, Daniel Cruz, Fabio Ferreira, Cleiton Tavares, and Eduardo Figueiredo. Github label embeddings. In *Proceedings of the 20th International Working Conference on Source Code Analysis and Manipulation*, pages 249–253. IEEE, 2020.

[18] Jailton Coelho, Marco Tulio Valente, Luciana L Silva, and Emad Shihab. Identifying unmaintained projects in github. In *Proceedings of the 12th International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, 2018.

[19] Gustavo Vale, Angelika Schmid, Alcemir Rodrigues Santos, Eduardo Santana De Almeida, and Sven Apel. On the relation between github communication activity and merge conflicts. *Empirical Software Engineering*, 25(1):402–433, 2020.

[20] Mary L McHugh. The chi-square test of independence. *Biochemia medica*, 23(2):143–149, 2013.

[21] Douglas Kramer. Api documentation from source code comments: a case study of javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153, 1999.

[22] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar 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.

[23] Janet Siegmund. Program comprehension: Past, present, and future. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 5, pages 13–20. IEEE, 2016.

[24] Asher Trockman, Keenen Cates, Mark Mozina, Tuan Nguyen, Christian Kästner, and Bogdan Vasilescu. Automatically assessing code understandability reanalyzed: combined metrics matter. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 314–318. ACM, 2018.

[25] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Proceedings of the International Symposium on Empirical Software Engineering*, pages 83–92. IEEE, 2004.