MASTER'S THESIS

CHARACTERIZING THE RELATION BETWEEN THE MAINTAINABILITY AND COMMUNITY ASPECTS OF OPEN SOURCE SOFTWARE PROJECTS

Duits, Jean-Pierre

Award date:
2024

CHARACTERIZING THE RELATION BETWEEN THE MAINTAINABILITY AND COMMUNITY ASPECTS OF OPEN SOURCE SOFTWARE PROJECTS

by

Jean-Pierre Duits

in partial fulfillment of the requirements for the degree of

Master of Science
in Software Engineering

at the Open University, faculty of Management, Science and Technology
Master Software Engineering
to be defended publicly on Thursday May 30th, 2024 at 10:30.

Student number:

Course code:	IM9906	
Thesis committee:	dr. Ebrahim Rahimi (chairman and first supervisor),	Open University
	dr. Bastiaan Heeren (second supervisor),	Open University

CONTENTS

ABSTRACT

A significant percentage of today's software is based on open source software (OSS). This includes not only small software projects but also large, popular, and widely used OSS projects such as operating systems (Linux) and web browsers (Firefox, Chromium). Often, OSS projects have built an evolving community with members around them, including developers, bug reporters, bug fixers, and various other roles. These members can join the project (magnetism) and can be very active or passive (stickiness). Based on these two dimensions of magnetism and stickiness, the popularity of a project can be determined and be categorized into the following categories: attractive, stagnant, fluctuating, or terminal. Maintainability is another important aspect of software, including OSS. It is a crucial component of software quality, for which numerous quality models are available. These quality models measure certain quality characteristics through various metrics. Although the quality as well as community aspects of OSS projects have been researched by other scholars, the influence of transitions between these four categories on the maintainability metrics of OSS projects remained under-explored. This study aims at filling this research gap.

We composed a quality model, adapted from existing models, that measures both quality (with a focus on maintainability) and the community related metrics of OSS projects. With this quality model, we attempt to investigate and characterize the possible relationship between the maintainability of software and changing popularity. For this, we used a custom created dataset consisting of 90 random Java projects from GitHub. To measure the metrics from the quality model we developed a tool to analyze the repositories in the dataset in multiple periods. Optionally, we tried to determine if it is possible to predict future maintainability characteristics based on current popularity and maintainability.

From our results, it appears that the maintainability characteristics of projects usually do not improve or worsen, even when the popularity of a project changes. Projects often change in popularity and seem to frequently end up in a stable phase without attracting new developers.

1

INTRODUCTION

Open source software (OSS) is software whose source code is publicly available. This source code may be modified, improved and distributed. Open source software is widely used for commercial and personal use. Some examples of well known OSS projects are the Apache webserver[1], the Linux operating system[2] and web browsers such as Firefox[3] and Chromium.[4]
An important difference between open source and closed source is the social aspect [13]. OSS projects often consists of a continuously changing community of users, readers, developers, bug reporters and bug fixers. These community members often do this voluntarily, causing frequent changes in the community structure.
The sustainable evolution of an OSS project strongly depends on the attraction (magnetism) for new developers and the extent to which they can be retained (stickiness). In a study by Yamashita et al., they linked this magnetism and stickiness to a number of categories that indicate the community state of an OSS project [24; 25]. A community state category can be attractive, fluctuating, stagnant or terminal.
Some popular platforms for hosting OSS projects are GitHub[5] and GitLab[6]. They offer all kinds of facilities for an OSS project such as Git version control, hosting, issue trackers and forums.
An important aspect of software, including OSS, is quality and in particular its maintainability as an important component of software quality which is the focus of this research. Software quality can be determined based on a number of metrics from a software quality model. Over the years, many quality models have been proposed, often with a focus on a specific topic, such as maintainability. The most well-known quality models are ISO 9126 [1] and its successor ISO/IEC 25010[7].
Much research has been done on the social aspects of OSS projects. For example: Ye and Kishida researched the motivation of OSS developers [26], Steinmacher et al. did research about how and why contributions from external developers were not accepted [21] and Torres et al. did an analysis of the core team role in an OSS Community. According to McClean et al., a gap in current studies is the changes in OSS quality over time and whether the structure of the team has an effect on its surrounding [13]. With our research, we will try to close

[1] https://httpd.apache.org/
[2] https://www.kernel.org/
[3] https://hg.mozilla.org/mozilla-central/
[4] https://www.chromium.org
[5] https://github.com
[6] https://gitlab.com
[7] https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en

this gap and gain insight whether the OSS quality of software depends on the structure of the OSS community, and what happens when this structure changes. To achieve this, we will answer the following main research question:

How can software maintainability of OSS community categories as defined by Yamashita et al. [24] be characterized?

For our research we will use the OSS community state categories (OSS community categories) of Yamashita et al. to classify the state of an OSS community [24; 25]. We will develop a tool that automatically measures these categories for selected OSS repositories in specific periods. Because Yamashita et al. do not measure the maintainability metrics of the software, we will do this during our research.
We will also build a prediction tool to predict the future maintainability, based on the current OSS state category and maintainability characteristics.
Therefore we will propose a software maintainability model, mainly based on existing models, with a focus on OSS by incorporating metrics that can represent the community and social aspects.
We did a literature study to compare existing quality models. An important aspect of the quality models should be that the metrics can be measured automatically.
Our research uses data from GitHub stored in a custom created dataset. To the best of our knowledge, no research has been done on the popularity of OSS communities and its influence on software quality. Our composite quality model and developed prediction model could be used by OSS communities in the future, and try to guarantee the quality.

The rest of the document is structured as follows: Chapter 2 describes all relevant studies and tools that are already available and that could possibly be used for our research. Topics such as OSS community structures (section 2.1), OSS population categories (section 2.2), software quality models (section 2.3) and code comments (section 2.4) are discussed here. The research methodology, including our formulated research questions (section 3.1), is discussed in chapter 3. In chapter 4 we compose our quality model and in chapter 5 we create a tool that uses this quality model for measuring our selected repositories and dataset. A characterization of maintainability in relation to OSS community categories is described in chapter 6, while chapter 6.5 describes a prediction tool for maintainability. Chapter 7 discusses the limitations and future work of our research.

2

RELATED WORK

Open source communities and software quality models are common research topics. As a result, a lot of information is already available for our research. In this section we describe the information that may be relevant to our research. In section 2.1 we summarize how an OSS community and the associated social structure is composed. To classify the status of an OSS project we can use OSS population state categories, we discuss this in section 2.2. Quality models that can be used to determine the quality of software are discussed in section 2.3.

2.1. OSS PROJECTS AND ITS SOCIAL STRUCTURE

In closed source software projects, developers and users are clearly defined and strictly separated. In open source software (OSS) there is no clear distinction between developers and users. All users are potential developers [26]. People involved in an OSS project create a community around the project bounded by their interest in using and developing the system. Members of an OSS community take on a role based on their own interests. Nakakoji et al. defined eight roles of OSS community members [15]. These roles are listed in table 2.1.

An OSS community has not a strict hierarchy, but it is neither flat. The influences that members have on the system and the community are different, depending on their role. Figure 2.1 shows how closer to the center, the larger the radius of influence [15].
Each OSS community has a unique structure depending on the nature of the system and its member population [26]. The roles and their associated influences in OSS communities can be realized only through contributions to the community. Roles are not fixed. Members can play larger roles if they aspire and make appropriate contributions [26]. As members change the roles they play, they also change the social dynamics and thus reshape the structure of the community, resulting in an evolution of the community itself.
Open source projects are usually initiated by a single developer, such as Linus Torvalds for the Linux project, by a core team of developers, such as the development of the Python programming language [13]. Developers can view the source code and contribute to it. While contributing to open source software, a lot of data is created about how developers work and interact together in a community [13]. A Social Network Analysis (SNA) can provide an overview of how the network of an OSS community is structured [13]. A necessary dataset for SNA is a set of connections between people. To obtain this data, a commit history can be analyzed. If people work on the same project at the same time, they might be said to be

Role	Description
Passive User	Users that just use the system, in general most members of a community are passive users [26].
Reader	Active users of the system, they not only use the system, but also try to understand how the system works by reading the source code.
Bug Reporter	Discovers en reports bugs.
Bug Fixer	Fix bugs, have to read and understand a small portion of the source code.
Peripheral Developer	Contribute occasionally (irregular) new functionality or features to the system.
Active Developer	Contribute regular new functionality, features or bug fixes.
Core Member	Responsible for guiding and coordinating the development of the project. Have been involved with the project for a relative long time and have made significant contributions to the development and evolution of the system.
Project Leader	Often the person who has initiated the project. Responsible for the vision and overall direction of the project.

Table 2.1: Roles of open source community members [15]

working together. If people comment on the same issue, there is also a link. Modern OSS development sites (such as GitHub) have additional social network features such as following or liking, which can be used by a SNA [13].

2.2. OSS COMMUNITY STATE CATEGORIES (OSS COMMUNITY CATEGORIES)

Onoue et al. have investigated what the population state of an open source community is [17]. To become and remain a successful open source project, it is important to attract new developers and keep these developers making contributions. A project is magnet when it attract a large proportion of new developers and sticky when it retains a large proportion of active developers [24]. We consider a developer as someone who writes or modifies code for a project. Yamashita et al. have studied the typical values of sticky- and magnet open source projects and analyzed how these values change over time. For the analysis they used the GitHub dataset of Gousios [1] [9]. This dataset contains a lot of code authorship data, such as commit and pull requests activities, about Github repositories and their evolution in time. Yamashita et al. defined four categories, based on the data of 90 open source projects, to classify OSS community categories. These categories are listed in table 2.2.

To calculate which OSS community category a project belongs to, at a given moment, there are two metrics. The magnet metric determines the number of new developers who have contributed to the project during a selected period. The sticky metric calculates which developers contributed previous period and current period. Sticky values considers only the number of contributors of previous period. These metrics are adopted from Yamashita et al.

[1] https://ghtorrent.org/

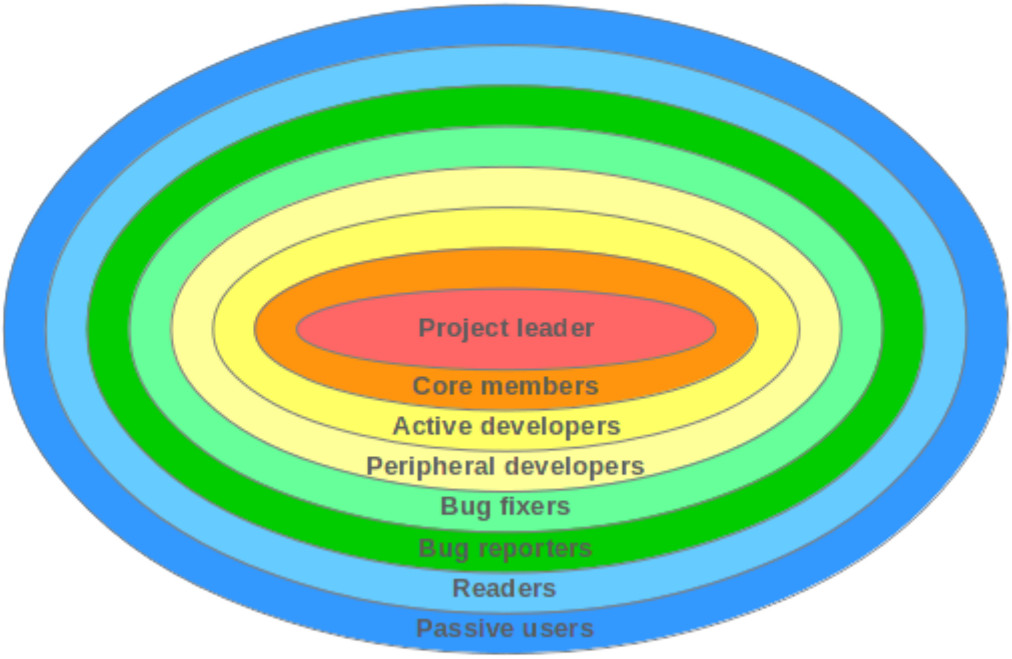


Figure 2.1: Structure of an OSS community [15]

Magnetism		
	Fluctuating Projects that successfully attracts new developers, but fail to retain them. High magnet, low sticky.	**Attractive** Projects that successfully attracts new developers and retain the existing ones. High sticky and high magnet.
	Terminal Projects that struggle to attract new developers and fail to retain them. Low magnet, low sticky.	**Stagnant** Projects that struggle to attract new developers, but successfully retain them. Low magnet, high sticky.
		Stickiness

Table 2.2: OSS community categories [24]

and adapted to the following formulas with P_{i-1} as period i:

$$\text{Magnet value} = \frac{\text{New developers in } P_i}{\text{Total developers}}$$

$$\text{Sticky value} = \frac{\text{Developers period } P_i\text{, with contributions in } P_{i-1}}{\text{Developers period } P_{i-1}}$$

The thresholds of the OSS community categories in table 2.2 are defined using the median of the sticky and magnet values [24; 25].
The examples in figure 2.2 show that during period P_i project 1 has 1 new developer (C) out of 3 total (A, B and C). Therefore the magnet value is: 1/3. In period P_{i-1} there were 2 contributing developers (A and B), of which 1 developer in P_i also made a contribution (A). Therefore the sticky value is: 1/2. For project 2 there are a total of 3 contributors (D, E and F) in period P_i, 2 of which are new (E and F). The magnet value is: 2/3. In P_{i-1} there is 1 contributor (D) that also contributes in P_i, so the sticky value is 1/1.

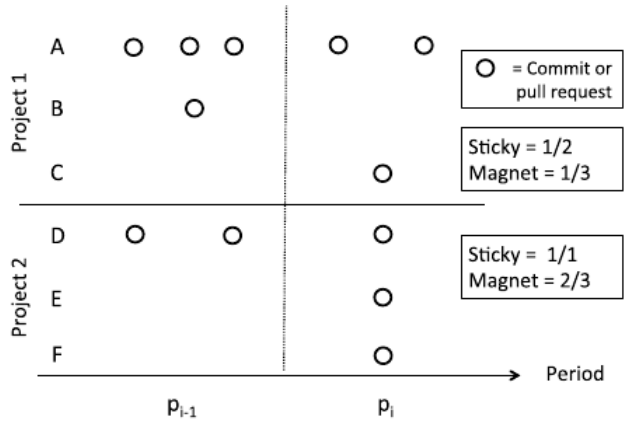


Figure 2.2: Calculation examples of sticky and magnet metrics [25].

2.3. SOFTWARE QUALITY MODELS

Over the past decades, many different quality models have been proposed to determine the quality of software systems. Quality models consist of a number of metrics with which the quality of software systems can be determined. The first quality model was described in 1977 by McCall et al. [12], but many models have been proposed over the years [5; 6]. Finally, software quality standards have been described by the International Organization for Standardization (ISO)[2].

2.3.1. MC CALL'S QUALITY MODEL

One of the best-known quality model is the model of McCall et al. [5]. This model originates from the US military, and it defines and identifies 3 major perspectives and 11 factors of the quality associated with 23 criteria of a software product [6; 12] (see figure 2.3). The 3 major perspectives are listed in table 2.3 [5].

Perspective	**Description**
Product revision	The ability of the product to apply changes. How complicated is it to fix bugs (maintainability), make necessary changes (flexibility) or test the system for errors and its specifications (testability).
Product operations	The functioning of the software product: does the software meet the specifications (correctness) and has it the ability not to fail (reliability). How efficiently does the software use the resources (efficiency), is it protected against unauthorized access (integrity) and how easy is the use of the software (usability).
Product transition	The adaptability of the product to new environments. Moving the software to another environment (portability), the ease of using the software in an other context (reuse-ability) and the effort to couple the system to another system (interoperability).

Table 2.3: Major perspectives of Mc Call's quality model [5]

[2] https://www.iso.org

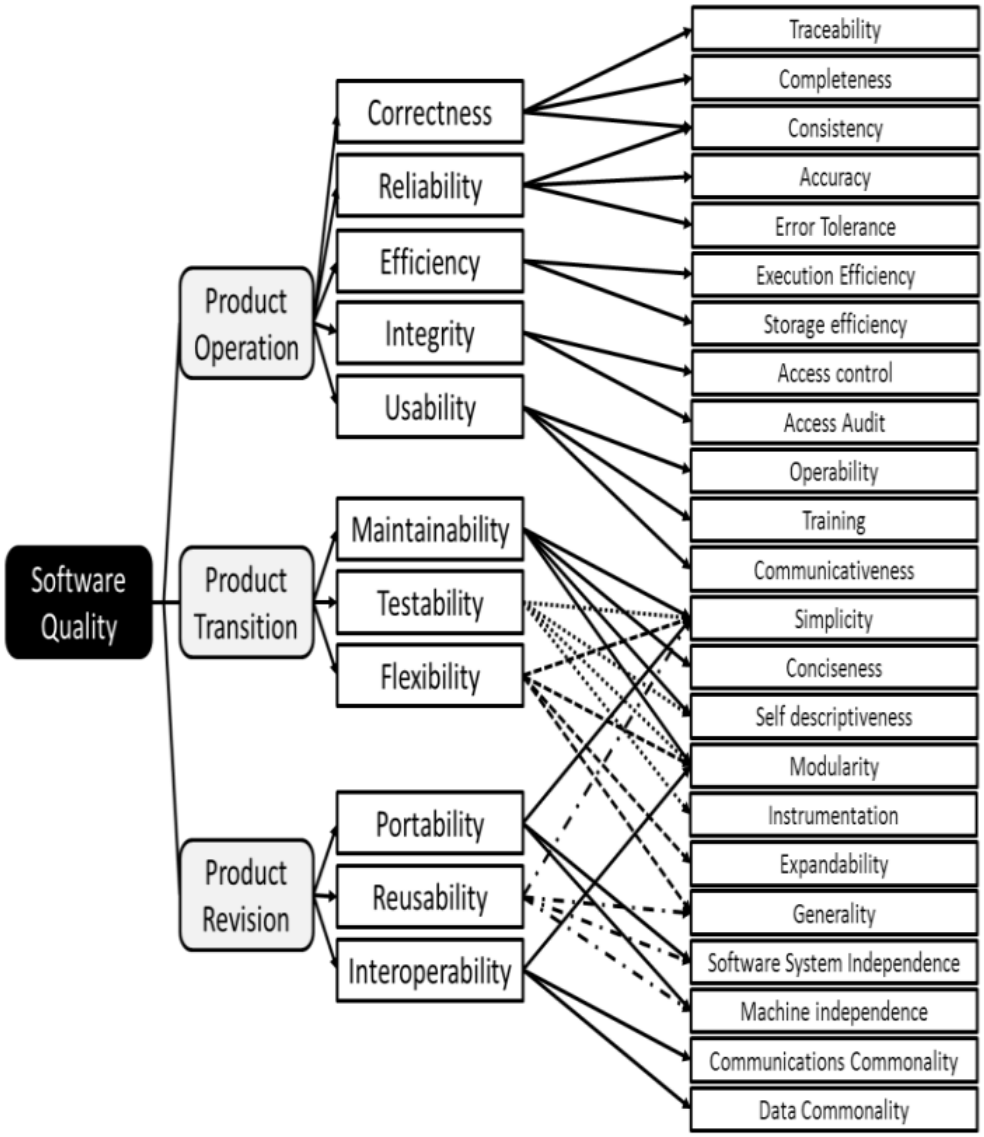


Figure 2.3: Mc Call's quality model [6]

2.3.2. ISO 9126 QUALITY MODEL

In 1991 a number of quality features, mainly derived from Mc Call's model, were published as the standard quality model [1] by the International Standards Organization (ISO) and named as ISO 9126 software quality model. The ISO 9126 model distinguishes three layers: internal quality (code quality without executing), external quality (during execution) and quality in use (user experience). Six main characteristics and 27 sub-characteristics are described [1; 10] (see figure 2.4). From 2001 to 2004, the ISO published an expanded version of the ISO 9126 quality model. This expanded version contains technical reports describing 16 metrics for external quality and 9 for internal quality [2–4]. The 6 main characteristics of the ISO 9126 quality model are listed in table 2.4.

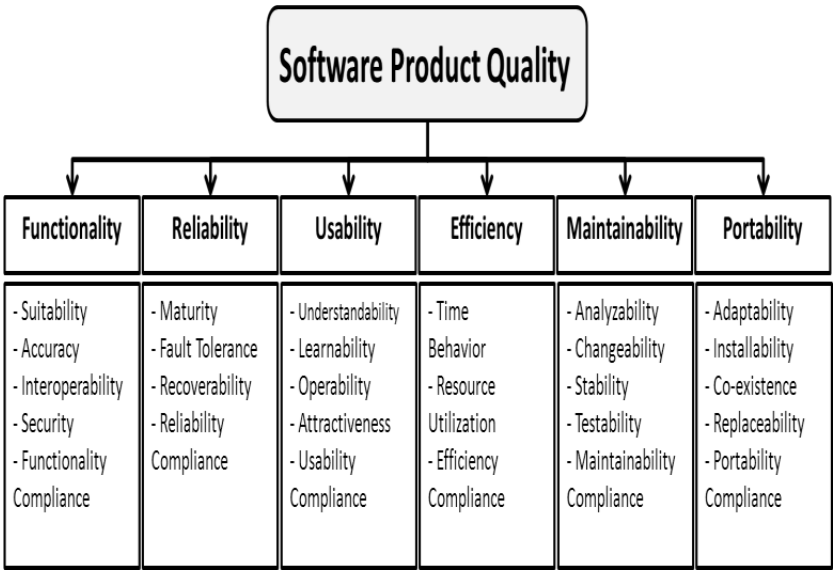


Figure 2.4: The main characteristics with the sub characteristics of the ISO 9126 quality model [1; 6]

2.3.3. ISO/IEC 25010 QUALITY MODEL

From 2011 there is a successor to the ISO 9126 model, the ISO/IEC 25010 model. It is part of ISO/IEC 25000 series, which is also called the SQuaRE model (System and Software Quality Requirements and Evaluation). The ISO/IEC 25010 extends ISO 9126. The ISO/IEC 25010 describes 8 main characteristics and 31 sub-characteristics (see figure 2.5)[3]. The main differences between the ISO 9126 quality model and ISO/IEC 25010 quality model are the introduction of 2 new main characteristics: security and compatibility. The 8 main characteristics of the ISO/IEC 25010 quality model are listed in table 2.4.

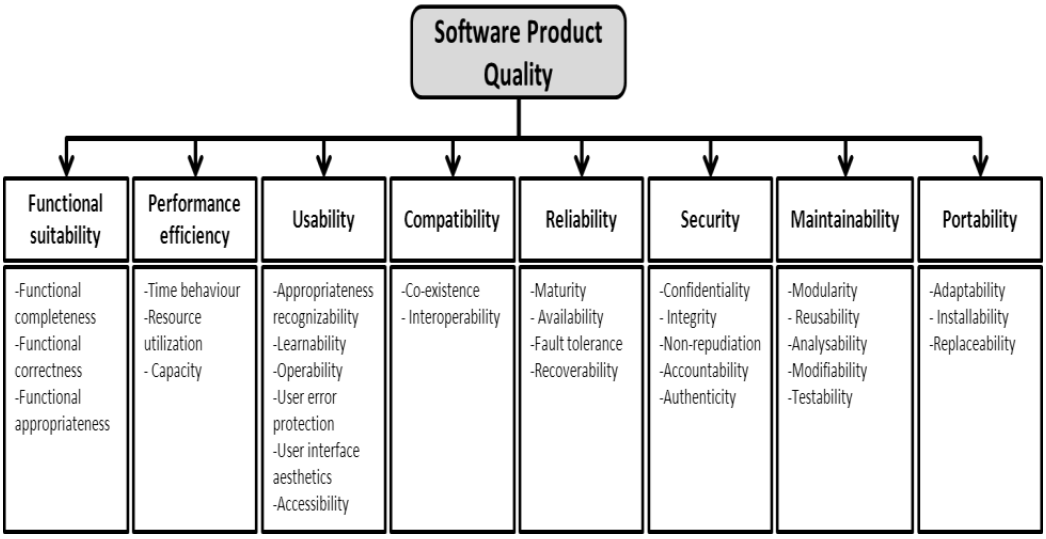


Figure 2.5: The main characteristics with the sub characteristics of the ISO/IEC 25010 quality model [1; 6]

[3] https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en
[4] https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en:sec:4.2.1

Characteristics	ISO model(s)	Description
Functional suitability	ISO 9126 ISO/IEC 25010	Degree to which the software product or system provides functions that meet stated and implied needs when used under specified conditions.
Performance efficiency	ISO 9126 ISO/IEC 25010	Performance relative to the amount of resources used under stated conditions.
Usability	ISO 9126 ISO/IEC 25010	Degree to which a product or system can be used by specified users to achieve goals with effectiveness, efficiency, and satisfaction in a specified context of use.
Compatibility	ISO/IEC 25010	Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware and software environment.
Reliability	ISO 9126 ISO/IEC 25010	Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time.
Maintainability	ISO 9126 ISO/IEC 25010	Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers.
Security	ISO/IEC 25010	Degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization.
Portability	ISO 9126 ISO/IEC 25010	Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another.

Table 2.4: Main characteristics of the ISO 9126 [1] and ISO/IEC 25010 quality models [4]

Limitation	Description
Analyzability	How easy or difficult is it to diagnose the system for deficiencies or to identify the parts that need to be modified?
Changeability	How easy or difficult is it to make adaptations to the system?
Stability	How easy or difficult is it to keep the system in a consistent state during modification?
Testability	How easy or difficult is it to test the system after modification?
Maintainability compliance	How easy or difficult is it for the system to comply with standards or conventions regarding maintainability?

Table 2.5: Maintainability characteristics of ISO 9126, adopted by the SIG maintainability model [10]

2.3.4. SIG MAINTAINABILITY MODEL

Many different models have been proposed over the years, often based on one of the ISO models. A well-known alternative model is the Software Improvement Group (SIG)[5] model, developed by Heitlager et al. [10]. This model is based on the ISO 9126 model and focuses on the maintainability characteristics. These characteristic are subdivided into five sub-characteristics, as listed in table 2.5.

The ISO 9126 model does not directly measure maintainability through source code or documentation, but rather indirectly through the activities of technical staff [10]. For this reason, alternative metrics, such as the maintainability index and the SIG maintainability model have been developed. The maintainability index (MI) was proposed by Coleman et al. and Oman and Hagemeister to objectively determine the maintainability of a software system based on the status of the corresponding source code [8; 10; 16]. The MI is a composite number on several metrics: The Halstead volume metric (HV), cyclomatic complexity (CC), the average number of lines of code per module (LOC) and the the percent of comment lines per module (COM). The higher the MI, the better the maintainability of the system. Calculating the MI is possible with the following formula:

$$171 - 5.2ln(HV) - 0.23CC - 16.2ln(LOC) + 50.0sin\sqrt{2.46 * COM}$$

However, Heitlager et al. argue that, based on their years of consultancy experience, there are a number of limitations to the MI and they believe it is not clear what steps should be taken to increase the MI [10]. The limitations they mention are the average complexity, the Halstead volume metric, comments, formula understandability and control [10]. These limitations are summarized in table 2.6.

Heitlager et al. have proposed an alternative maintainability model based on these limitations, the SIG maintainability model. It consists of a set of source-code measures that are ranked with: $++$ / $+$ / o / $-$ / $--$, and mapped to the maintainability sub-characteristics [10]. The characteristics that are described are volume, complexity per unit, duplication, unit size, and unit testing. They will be described in the following paragraphs. A source code unit is the smallest piece of code that can be executed, such as methods and procedures.

VOLUME

A larger the system requires a larger effort to maintain. Lines of code (loc): All lines of the source code that are not comments or blank lines. Volume can be ranked using the values

[5] https://www.softwareimprovementgroup.com/

Limitation	Description
Root-cause analysis	Since the MI is a composite number, it is very difficult to determine what causes a particular value for the MI.
Average complexity	Using the average cyclomatic complexity (CC) metric for composing the MI tends to mask the presence of risky parts.
Computability	The Halstead Volume metric (HV) is not widely accepted within the software engineering community and is difficult to compute.
Comments	More often comments are code that has been commented out or natural language referring to earlier version of the code. Sometimes it describes a complex piece of code that is difficult to maintain. Therfore counting comments has no relation to maintainability.
Understandability	It is unclear what the constants and variables from the MI formula are based on.
Control	It is unclear to both developers and management what needs to be done to improve the MI.

Table 2.6: Limitations of the MI according to Heitlager et al. [10]

from table 2.7. A project with fewer than 66 kloc of code will be ranked as ++, a project with more than 1310 kloc will be ranked as −−.

Rank	Kloc
++	0-66
+	66-246
o	246-665
−	665-1310
−−	> 1310

Table 2.7: Ranking of kloc [10]

Complexity per unit

In 1976 McCabe introduces the concept of Cyclomatic Complexity (CC), which is a measure of software code complexity [11]. This metric is designed to help software developers to analyze the complexity of a program's control flow. CC measures the number of linearly independent paths and is calculated by analyzing the control flow graph of a program. It takes into account the number of decision points (such as if statements, loops, and case statements) in the code. The higher the CC value, the more complex the code is. Complex code indicates a potentially greater risk for bugs. The formula for CC is given below, where $v(G)$ is the cyclomatic complexity of the graph G, e is the number of edges and n is the number of vertices.

$$v(G) = e - n + 2$$

Pressman described an example for calculating the CC [18]. In the source code example in Figure 2.6, each node in the graph of Figure 2.7 is referenced. The CC can be calculated with

$$v(G) = 17 - 13 + 2 = 6$$

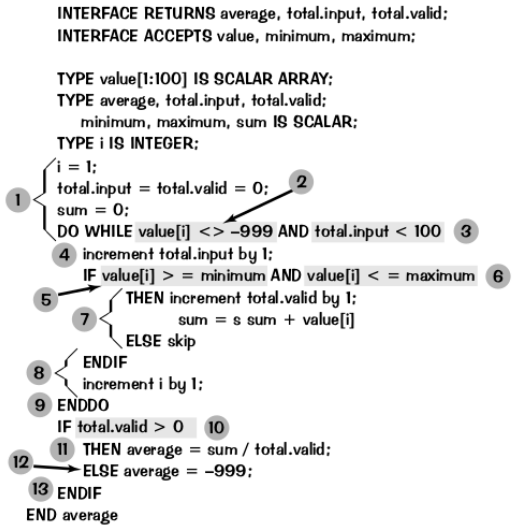


Figure 2.6: Source code with nodes identified [18]

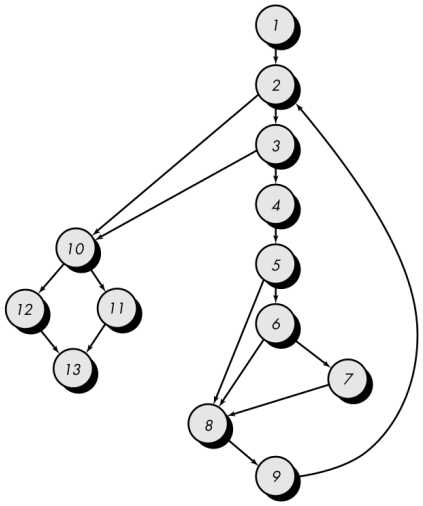


Figure 2.7: Flow graph for Figure 2.6 [18]

Complex units are difficult to understand (analyzabilty) and difficult to test (testability). The SIG maintainability model adopted this metric and calculates for each unit the CC and categorize them by risk as listed in table 2.8.

CC	Risk Evaluation
$1-10$	Simple, without much risk
$11-20$	More complex, moderate risk
$21-50$	Complex, high risk
>50	Very complex, very high risk

Table 2.8: Risk evaluation of CC per unit [10]

The ranking can be determined based on the collected risks of complexity. This ranking can be determined using the listing in table 2.9. It is important that all percentages meet the thresholds for the respective ranking. When the CC is moderate in maximum 21% of the source code units, and there are no high or very high units, the ranking is ++.

	Maximum Relative loc		
Rank	**Moderate**	**High**	**Very High**
++	21%	0%	0%
+	30%	5%	0%
o	40%	10%	0%
−	50%	15%	5%
−−	-	-	-

Table 2.9: Ranking of CC per unit [10]

DUPLICATION

Duplication of source code fragments (code clones). Excessive duplication makes a system larger than it needs to be. Duplicated blocks over 6 lines, ignoring leading spaces, are calculated as code duplication. A well designed system should not have a percentage of more

than 5% code duplication. Based on the percentage of duplicates, a ranking can be assigned, as shown in table 2.10.

Rank	Duplication
++	0-3%
+	3-5%
o	5-10%
−	10-20%
−−	20-100%

Table 2.10: Ranking duplication [10]

UNIT SIZE

The size of an source code unit does say something about the maintainability. Larger units are more difficult to maintain (lower analysability and testability). Unit size uses a simple loc metric. Methods with fewer than 30 lines of code have a low risk. If a method has more than 74 lines of code, it has a very high risk. Refer to table 2.11 for a comprehensive list of risk evaluations for unit sizes.

Unit Size (loc)	Risk Evaluation
$0-30$	Low risk
$30-44$	Moderate risk
$44-74$	High risk
>74	Very high risk

Table 2.11: Risk evaluation of unit size [10]

The ranking can be determined by the percentages of risks as listed in table 2.12. It is important that all percentages meet the thresholds for the respective ranking. When 19.5% or lower of the code has a moderate risk, less than 11% has a high risk, and less than 4% has a very high risk, then the unit size is ranked as ++.

	Maximum Relative Unit size		
Rank	Moderate	High	Very High
++	19.5%	10.9%	3.9%
+	26%	15.5%	6.5%
o	34.1%	22.2%	11%
−	45.9%	31.4%	18.1%
−−	-	-	-

Table 2.12: Ranking unit size [10]

UNIT TESTING

Good sets of unit tests have a significant positive impact on maintainability. Unit test coverage can be measured with dedicated tools. The test coverage can be ranked as listed in table 2.13.

Rank	Unit Test Coverage
++	95-100%
+	80-95%
o	60-80%
−	20-60%
−−	0-20%

Table 2.13: Ranking of unit tests [10]

MAPPING BACK TO SYSTEM LEVEL

The obtained rankings of the described metrics can be mapped back to the maintainability characteristics of the ISO 9126 model. The rows in table 4.3 represent the 4 characteristics of the ISO 9126 model, while the columns represent the source code properties of the SIG maintainability model. To calculate a system-level value, the average is computed from the relevant properties for this system-level score. The relevant properties are indicated with a cross. In table 2.15, an example of a mapping of obtained rankings to a system level score is provided.

	Volume	**Cyclomatic complexity**	**Duplication**	**Unit size**	**Unit testing**
Analysability	x		x	x	x
Changeability		x	x		
Stability					x
Testability		x		x	x

Table 2.14: Mapping back to maintainability characteristics of the ISO 9126 model [10]

2.4. CODE COMMENTS

Comments in source code can be used to explain the programmer's intent or to summarize the code [14]. Misra et al. investigated the co-relation between quality of code comments and issues. For this study, they created a machine learning model that can categorize code comments into two different categories: relevant comments and auxiliary comments. Relevant comments are considered useful, while auxiliary comments do not add value for developers. Comments consist of various types. Steidl et al. defined these types [20]. Misra et al. completed the list with 'API or IDE comments' [14]. Relevant comments include the types as listed below:

- **Header comments** Describes the functionality of a class. Header comments are located before the class declaration.
- **Member comments** Information about the method or field. Located before or in the same line of the definition.

	Volume	Cyclomatic complexity	Duplication	Unit size	Unit testing	Score
Obtained ranking	++	–	-	-	o	
Analysability	x		x	x	x	**o**
Changeability		x	x			**-**
Stability					x	**o**
Testability		x		x	x	**-**

Table 2.15: Mapping example

- **Inline comments** Implementation decisions within a method.
- **Section comments** Groups multiple methods or fields together. For example: `// -- Getter and Setter Methods --`
- **Task comments** Notes or remarks of the developer. For example a TODO note or an implementation explanation.
- **Code comments** Commented out code, so it is ignored by the compiler.
- **API or IDE comments** Comments that contain all API calls and IDE directives.

For auxiliary comments, the following types are described:

- **Copyright comments** Information about the license and the copyright of the source code. Mostly placed on the top of the file.
- **Noisy comments** Comments without contributing valuable information.

Misra et al. extracted the comments and issue metadata, such as opening and closing times, from 625 Python repositories on GitHub. With this collected meta data, they calculated the average duration time for resolving an issue. The code comments of the repositories were classified into relevant comments or auxiliary comments using a trained machine learning model (Random Forest model). With all this data, Misra et al. investigated how the average time for resolving an issue varies with the number of relevant comments. They have categorized the results into four categories, as listed in table 2.16.

	Relevant comments	Average time taken to solve an issue
1	Low percentage	More time
2	Low percentage	Less time
3	High percentage	More time
4	High percentage	Less time

Table 2.16: Repository density categories [14]

The graph in Figure 2.8 is divided into the 4 described categories from table 2.16. The categories are separated by the mean. For the x-axis representing the percentage of relevant

comments, it is 0.88 (88%). For the y-axis representing the mean number of days to resolve an issue, it is 97 (days). The largest number of repositories in Figure 2.8 is in category 4. According to Misra et al. is that the evidence that source code with more relevant comments resolves issues faster. The result indicates that when the percentage of relevant comments in a repository is higher than 88%, the average time to resolve an issue is less than 100 days in 65% of cases [14].

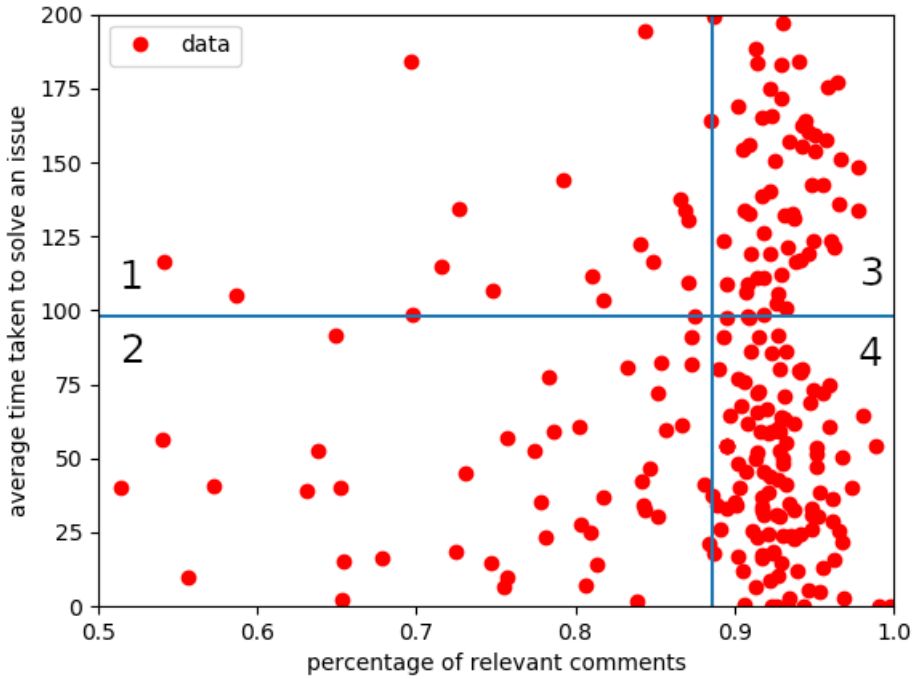


Figure 2.8: Percentage of relevant comments vs average time to solve an issue [14]

3

RESEARCH

In our research we want to investigate the possible relationship between the OSS community categories of Yamashita et al.: attractive, fluctuating, stagnant and terminal [24] (section 2.2) and their maintainability metrics (section 2.3.4).

3.1. RESEARCH QUESTIONS

For our research we define a number of research questions. These consist of a main question followed by 4 sub-questions.

3.1.1. MRQ: HOW CAN SOFTWARE MAINTAINABILITY OF OSS COMMUNITY CATEGORIES AS DEFINED BY YAMASHITA ET AL. [24] BE CHARACTERIZED?

We aim to understand the popularity of a repository by examining OSS community categories and assessing their impact on the maintainability. We do this by measuring 8 different periods for 90 repositories in our dataset, using our quality model from sub question 1 (section 4). With these measurements, we can analyze and characterize if there is a correlation between maintainability and an OSS community category.

3.1.2. SQ1: HOW CAN WE DEFINE OR COMPOSE A COMMUNITY-BASED OSS QUALITY MODEL THAT MEETS THE CURRENT STANDARDS?

To assess the quality of OSS, a quality model must be defined. We can use this model throughout the research. We have to do a literature study about existing quality models. This study allows us to assess the advantages and disadvantages of already existing models. An important aspect of the quality model should be that you can measure without, or with minimal, human interaction. This means that the measurements of the software projects can be fully automated. Another important parts are the properties to be measured. Due to time constraints, our research will focus on the maintainability part of the software. The maintainability part is mainly based on the source code of the software and therefore relatively easy to measure automatically. Properties of the OS community are also important and are therefore also included in the measurements. Many of these properties can be filtered out of our dataset (section 3.2). In chapter 4, the entire quality model is described in detail.

3.1.3. SQ2: How can we determine maintainability for the selected OSS projects?

To measure the quality based on our defined model from SQ1, we develop a command line (CLI) tool. This tool should measure a selected project from our dataset within a specific period. The tool should measure both the community and maintainability aspects. To measure the community aspects, we will need to use the GitHub metadata from the dataset. For maintainability, the selected repository will also be checked out using Git to analyze and measure the source code. The tool returns all measured results in a processable format, such as CSV or JSON. Refer to chapter 5 for the detailed explanation of this research question.

3.1.4. SQ3: How does software maintainability change when open source software projects make transitions between OSS community categories?

In our dataset, there are various periods that we can measure. Between these periods, the popularity can increase or decrease, causing the OSS community category to change. We are investigating whether we see a relationship between a transition of OSS community categories and the maintainability of a repository.

3.1.5. SQ4: How can we propose a predicting model for OSS software maintainability? (optional)

For this optional research question, we attempt to predict, with the help of machine learning, what the maintainability will be when a repository makes a transition to a different OSS community category. We develop a small tool where you can specify an OSS community category including the current maintainability rankings, and this tool returns the 4 possible transitions with the predicted new maintainability rankings.

3.2. Preparing dataset

During the research we use a custom created dataset. This dataset should be as varied as possible, and should therefore include popular and less popular Java projects from GitHub. We do not use the well known GHTorrent dataset of Gousios [9]. This dataset is outdated and is no longer maintained. The related website is offline and the last public GHTorrent dataset dates from 2019. Attempts to get the outdated software running (written in Ruby) have been unsuccessful. We decided to build our own tool that retrieves and stores all metadata from the GitHub API for selected projects in a database.

3.2.1. Git

Git [1] is a version control system for software systems, created by Linus Torvalds, the founder of Linux. Git makes it possible to track all changes in the source code during development. It enables developers to simultaneously work on different features in separate branches of a repository and merge these changes into a target branch. Changes in a branch that are saved in Git are called commits. Git ensures that developers can collaborate effectively. A developer works locally with a Git client to push and pull changes to or from a Git server. While there are alternative systems like Subversion (SVN) [2], is Git currently the most widely

[1] https://git-scm.com/
[2] https://subversion.apache.org/

used version control system.

3.2.2. GitHub

GitHub [3] is a platform based on Git to host repositories. Within this platform, various additional options have been added, such as issues, pull requests and wikis. Other notable platforms are GitLab [4] and SourceForge [5], but GitHub is currently the most popular for OSS communities.

3.2.3. Pull requests

GitHub users can clone existing OSS repositories, create a new branch, make changes to the code and submit a pull request to propose their changes to merge with the main/master repository. The project maintainers will review the changes and, if deemed valuable, merge them into the original master/main code base. GitHub offers three different ways to merge a pull request, the maintainer chooses which method to use. See table 3.1 for the GitHub merge methods.

Merge Method	Description
Merge commit	Creates a new commit to combine changes from the source branch into the target branch.
Squash and merge	Combines all the changes into a single, new commit before merging into the target branch.
Rebase and merge	Transfers the changes from the source branch to the target branch by replaying each commit. This results in a linear Git history.

Table 3.1: GitHub pull request merge methods

3.2.4. Selected OSS projects from GitHub

Github top lists contain a large number of popular tutorials and examples. So using this lists should not give a good representation of OSS projects. We choose to compose the project list ourselves via the public REST API of GitHub [6]. By calling the API with specific parameters we get a filtered list of public GitHub repositories. We have filtered the repositories to be 50MB or less in size and exist from at least 2020 with a Git push in the last year. To get a varied dataset we requested a list of repositories 3 times. Popular projects with more than 9000 stars, less popular projects with 1001..3000 stars, and low projects with less than 1000 stars. We have hand-filtered these lists by language (readable, no Chinese), no manuals, and the programming language in the repository is at least 90% Java. As a result, we have a diverse list of Java repositories, see appendix 7.5 for a complete list of the selected repositories.

3.2.5. Mining GitHub

Now we have a list of selected repositories, we can start mining them from GitHub and storing the data in the MySQL database. We have built a command line tool (CLI) for this using Laravel Zero, a PHP framework for building CLI apps. By using various endpoints of the

[3] https://www.github.com
[4] https://www.gitlab.com
[5] https://sourceforge.net/
[6] https://docs.github.com/en/rest

GitHub API, we can gather a lot of data about the repositories. If a rate limit is reached (5000 requests a hour), the tool waits until the time limit is exceeded. The tool also ensures that data is not redundant, for example by storing the same user only once. Running the tool can take many hours. The retrieved data is described below.

Users All GitHub parts can be linked to users. When a linked user does not yet exist in the database, it is created. The most relevant data stored for a user: GitHub ID, login name, full name and email address.

Repository Includes the GitHub ID, the repository name, the owner user, the default branch, whether it is a fork, and the date when the repository was created/modified/pushed.

Commits All the metadata of Git commits linked to the repository default branch or to a fork of a pull request. The most relevant metadata of these commits that is stored includes: SHA hash, author, committer, commit message, commit date and parent commit.

Stargazers A repository can have a number of stargazers (stars), which are a kind of likes on GitHub from users. The most relevant data for stargazers includes: starred at date and linked user.

Issues Issues are a specific component of GitHub. An issue is linked to a user, has a state (open or closed), create date, close date, title, body, comments, and labels. These linked labels, such as bug or question, are also stored in the database. This allows, for example, the determination of the number of bugs or feature requests for a specific period.

Forks Forks are repositories that are cloned from the original repository. These can be used, for example, to make modifications and create a pull request. Relevant data stored about forks includes GitHub ID, owner, name, parent repository and creation date.

Pull requests Requests from developers to merge changes into the original code base. Relevant data for pull requests includes GitHub ID, number, state (merged), user, creation date, close date, merge date, merge commit SHA, head user, head repository, head SHA, base user, base repository, and base SHA.

See Appendix B for the complete database structure including all stored fields. After fetching all the data from our selected repositories (section 3.2.4), The amount of data, as shown in table 3.2, is available.

Data	Count
Commits	272k
Forks	166k
Issues	124k
Labels	1.2k
Pull Requests	49k
Repositories	9k
Stargazers	648k
Users	414k

Table 3.2: Record count dataset

3.2.6. VALIDATION

The tool we have developed retrieved metadata for the selected repositories (section 3.2.4) from GitHub (section 3.2.2) via the public GitHub Rest API[7] and stored it in a database. To ensure the accuracy of this data, we randomly select projects and manually compare them for validation. This manual comparison can involve analyzing the repository on the GitHub webpage[8] or manually requesting the GitHub Rest API using tools such as Postman [9]. As the data was collected some time ago, we should consider that the data may have undergone some changes. For example, forks may have been removed, and stars could have been unstarred during the period between data retrieval and validation.
Per repository, we collect a range of metadata, such as details about forks, issues, commits, pull requests, and stargazers. Not all of this data is currently used but may be useful for future research.

Forks For the repository "exchange-core" [10], the GitHub API returns 705 forks, out of which 46 are from a date after our data import. Therefore, our dataset should have 659 forks. However, it currently shows 666 forks. After a manual comparison, it was discovered that 7 forks had been removed after the data import. This indicates that the data in our database is accurate.

Issues The database has recorded 136 issues for the "exchange-core" repository. The GitHub API returns 138 issues, of which 2 have a date after the data import. This confirms that the data in the database is accurate.

Commits A call to the GitHub API returns 293 commits for the "exchange-core" project. The database contains 290 commits. The three commits that do not match have the message: "This commit does not belong to any branch on this repository and may belong to a fork outside of the repository." We consider the data in the database as valid.

Pull requests Our database contains 43 pull request records for the "exchange-core" repository, and the GitHub API also returns 43 pull requests. We have compared the IDs, and they match. The pull requests are thus valid. Each pull request is associated with commits, and we are comparing the commits of 5 random pull requests via the GitHub repository page [11]. Commits always have an ‘author_id‘ and a ‘committer_id‘, so the user must exist in the database.

Pull request no.	Commits on GitHub	Commits DB	Matching commits SHA
102	4	4	Yes
134	4	4	Yes
92	1	1	Yes
29	22	22	Yes
2	3	3	Yes

Table 3.3: Random pull request commits comparison for the exchange-core repository

[7] https://docs.github.com/en/rest
[8] https://www.github.com
[9] https://www.postman.com/
[10] https://github.com/exchange-core/exchange-core
[11] https://github.com/exchange-core/exchange-core/

Stargazers The number of stargazers in the database for the "exchange-core" repository is 1628. However, the GitHub API returns 1612 stargazers for the same period, resulting in a difference of 16 stargazers. Because the missing stargazers are also not present in the API response, it is likely due to users un-starring the repository.

4

COMPOSING A COMMUNITY-BASED QUALITY MODEL

In this chapter, we address sub question 1: *How can we define or compose a community-based OSS quality model that meets the current standards?*. We answer this question by composing a quality model that we can use to answer the remaining research questions.

This model includes community and maintainability aspects. Throughout the research, we examine the relationship between the community and maintainability aspects. Our quality model is composed of various existing quality models. For the maintainability part, we have added a custom metric.s

4.1. COMMUNITY

Fundamental elements of an OSS community included popularity, commitment and contribution frequency. To measure this, we use the sticky and magnet metrics proposed by Yamashita et al. [24] (section 2.2), applying the following formulas:

$$\text{Sticky value} = \frac{\text{Developers period } P_i\text{, with contributions in } P_{i-1}}{\text{Developers period } P_{i-1}}$$

$$\text{Magnet value} = \frac{\text{New developers in } P_i}{\text{Total developers}}$$

Based on the stickiness and magnetism values, we can determine an OSS community state category. We refer to this category in our model as the OSS community category. The threshold to determine whether a value is high or low is the median of all sticky or magnet measurements. By using these thresholds, the OSS community category in which the project falls can be determined by using table 2.2.

4.2. MAINTAINABILITY

To assess the maintainability of an OSS repository, we use the SIG maintainability model of Heitlager et al. [10] as a foundation, (section 2.3.4). We extended this model and introduce

an additional software property related to comments in the source code. The original maintainability index (MI) of Coleman et al. [7], for which the SIG maintainability model is an alternative, has an optional feature to include the percentage of comments in the calculations. We reintroduce this aspect to our custom model. Our extended SIG maintainability model consists of the following metrics:

Volume This metric, assigns a ranking based on the lines of code, excluding comments and blank lines. The ranking is determined based on table 2.7 (section 2.3.4).

Complexity per unit For each unit (method) the cyclomatic complexity will be calculated and assigned to a risk as listed in table 2.8. Then, a ranking will be assigned based on the percentage of relative lines of code per risk. These rankings are listed in table 2.9 (section 2.3.4).

Duplication Duplicated blocks exceeding 6 lines, excluding leading spaces, are identified as code duplication. By determining the percentage of duplicates relative to the total lines of code, we can assign a ranking based on table 2.10 (section 2.3.4).

Unit size Based on the number of lines for each unit, a risk is assigned as listed in table 2.11. Then, a ranking will be assigned based on the percentage of relative lines of code per risk. These rankings are listed in table 2.12 (section 2.3.4).

Unit testing The SIG maintainability model has a metric to rank test coverage, as listed in table 2.13. Since it involves a procedure where the software must be built and various manual adjustments need to be made, it is error-prone for a large number of repositories, as in our research. Therefore, this metric is not included in our model. When determining the maintainability characteristics, this metric is replaced with the comments metric below.

Comments Comments in code contribute to various aspects. They can enhance the readability of the code and clarify the structure of the code to the developer or other members of the OSS community. As a result, comments contribute to both changeability and analyzability. The original maintainability index of Coleman et al. [7], for which the SIG maintainability model is an alternative, has an optional metric to include the percentage of comments in the total score of the maintainability index. The original MI has a maximum score of 171, which can be increased by 50 with an optional metric for comments. This additional score can be calculated using the following formula [7]:

$$\text{MI additional score} = 50 \cdot \sin\left(\sqrt{2.46 \cdot \text{PERCENTAGE\_COMMENTS}}\right)$$

PERCENTAGE_COMMENTS represents the percentage of all comment lines relative to the total number of lines. The formula returns a score for the percentage of comments in the code and has a maximum value of 50. A representation of the results of the formula is shown in the graph of figure 4.1. For example: with a comments percentage of 28% the score is 36.9, with 36% the score is 40.4. We have classified the scores that can be achieved into a ranking. These rankings are listed in table 4.1.

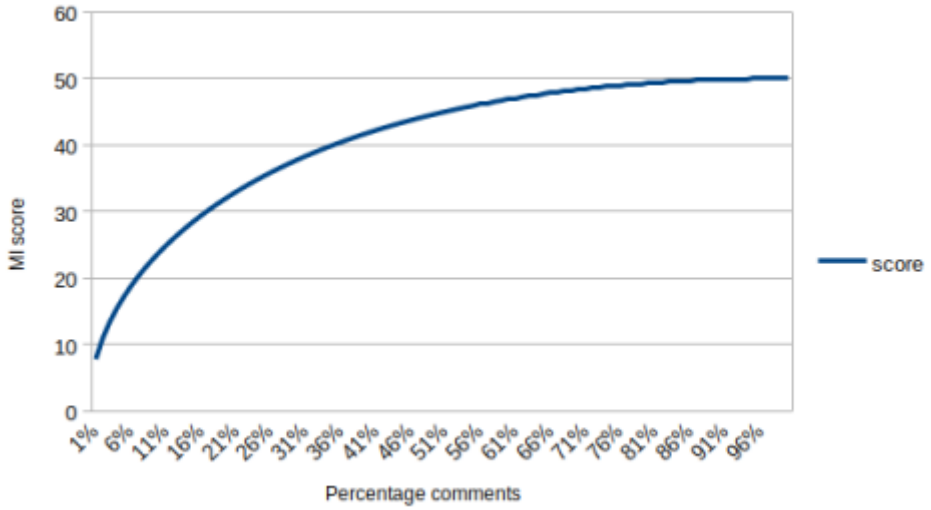


Figure 4.1: Percentage comments and their corresponding scores

Rank	MI score comments
++	40-50
+	30-40
o	20-30
−	10-20
−−	<10

Table 4.1: Ranking of comments

When we convert these rankings to the associated percentages, we obtain the table as shown in 4.2. The ++ ranking has a significant weight on the overall score. To mitigate this, we combined the metric from Misra et al. for this ranking [14]. According to Misra et al., the relevance of comments is important for the time required to resolve issues. This percentage is set at 88% relevant comments and is thus important to achieve a ++ ranking. As a result, we not only consider the quantity of comments but also their quality. For example: a percentage of comments of 35% (or above) of which 88% (or above) are relevant, the score is ++. Otherwise it is + or lower (table 4.2).

	Percentage comments	
Rank	**Percentage comments total**	**Percentage relevant comments**
++	35%	88%
+	17%	-
o	7%	-
−	2%	-
−−	-	-

Table 4.2: Ranking of percentage comments

4.3. FINAL MAINTAINABILITY SCORE

Based on the metrics related to the community, we can measure how sticky and magnetic a repository is during a specific period. With these values, we can determine the OSS commu-

nity category. So a repository is: attractive, fluctuating, stagnant or terminal. For the maintainability part, we use the mapping table, as listed in table 4.3. There are 3 maintainability characteristics: analysability, changeability and testability. These characteristics have their own score based on the average of the metrics related to them. For example the outcome for changeability is: the average of the measurement outcomes of cyclomatic complexity, duplication, and comments. The original SIG maintainability model also has a maintainability characteristic for stability. This is not included in our research because it is only based on the unit test metric, which we do not use.

	Volume	**Cyclomatic complexity**	**Duplication**	**Unit size**	**Comments**
Analysability	x		x	x	x
Changeability		x	x		x
Testability		x		x	

Table 4.3: Mapping back the proposed maintainability model to maintainability characteristics of the ISO 9126 model

With this model, we can measure the popularity and maintainability for each OSS repository, making it suitable for answering our other research questions.

5

DETERMINING THE MAINTAINABILITY OF OSS COMMUNITY CATEGORIES

Using the quality model resulted from SQ1, we can answer sub question 2: *How can we determine maintainability for the selected OSS projects?*

We have developed a tool that can measure the described metrics from the quality model. The GitHub Data tool (GHD) uses our composed dataset with GitHub metadata (section 3.2). Because the use of a graphical interface is not necessary, we have chosen to develop a command line (CLI) tool. We developed GHD using the Laravel Zero framework [1] for PHP [2]. This framework is focused on the development of CLI tools. The source code of GHD is available on GitHub [3]. When GHD is executed with specific arguments and options, a selected repository within a specific period can be measured. During measurement, GHD uses various external tools, such as Cloc, PMD and Simian, to support in calculating various metrics. The results can be returned in CSV, JSON, or as terminal output. All results are also stored in a MySQL database. It is possible to provide an UUID (universally unique identifier, consisting of 32 characters) as an option. With these UUID it is possible to trace back the related results in de MySQL database for the current GHD run. The execution of GHD proceeds as below. The owner and the repository should exist in the dataset:

```
ghd get:project-state <owner> <repository> [options]
```

Options:

```
--run-uuid[=RUN-UUID]              Run ID (UUID)
--start-date[=START-DATE]          Starting date (YYYY-MM-DD)
--end-date[=END-DATE]              End date (YYYY-MM-DD), default today
--interval[=INTERVAL]              Interval (week(s), default 26) [default: "26"]
--output-format[=OUTPUT-FORMAT]    Output format (json, csv or cli) [default: "cli"]
```

The execution example below measures the project Strassen-matrix-multiplication-Java in our dataset, during the default period of 26 weeks (half a year) starting at 01-01-2019 until 02-07-2019. The results are saved as a CSV file.

[1] https://laravel-zero.com/
[2] https://www.php.net
[3] https://github.com/jpduits/ghd

```
ghd get:project-state MoeidHeidari Strassen-matrix-multiplication-Java
 --start-date=2019-01-01 --output-format=csv
```

When we create a bash script that iterates through all 90 repositories in our dataset, we execute GHD for every repository and obtain a complete set of results as output and in our database. During the execution of GHD, various measurements are carried out on the selected repository. When a start date, interval, and end date are provided, multiple periods are measured between this start and end date. These measurements relate to community and maintainability aspects or metrics. Both will be further described in section 5.1 and section 5.2. For each repository, the data for each measured period is stored as shown in table 5.9.

5.1. MEASURING THE MAINTAINABILITY OF OSS PROJECTS

To determine maintainability, GHD needs the source code available that was relevant at the selected period. Therefore, it first clones the respective repository using Git. This is done in a temporary directory. GHD checks out on a commit of the default branch (mostly the main or master branch) that is closest to the end date and not older than the start date of the period. If there is no commit available in this period it takes the last commit before the start date. With the source code now available, GHD can perform the measurements as described below. It measures only the .java files and excludes files related to (unit) tests. GHD generates a file list that can be used by various metrics so all metrics and external tools work with the same files. Determining lines of code (LOC) can be done in various ways, such as using NCSS [4], for example. Since not all tools support all LOC methods, GHD uses the most common approach: all lines of code except comments and blank lines, as suggested by Heitlager et al. [10].

Volume To calculate the volume of a repository, GHD uses the cloc tool [5]. Cloc returns the number of blank lines, comment lines, and code lines for the file list we have defined. Based on the number of lines of source code, GHD can apply the SIG ranking as listed in table 2.7.

Unit size A unit is the smallest piece of code that can be executed and tested [10]. In our measurements, we consider methods and constructors as units. The LOC of a unit can be measured using the Checkstyle tool [6]. This tool allows GHD to retrieve the LOC for all methods and constructors in our files from the file list and assign a risk to them, as listed in table 2.11. When each method has been assigned a risk, GHD can calculate the relative percentages based on the total LOC and classify them according to the SIG ranking from table 2.12. The LOC of each unit are also used for the complexity per unit metric.

Complexity per unit To calculate the cyclomatic complexity of all methods in the files from the file list, GHD uses the PMD tool [7]. PMD uses the formula $v(G) = e - n + 2$ as described in 2.3.4 [8]. GHD calculates the cyclomatic complexity for each method and

[4] https://docs.pmd-code.org/pmd-doc-6.21.0/pmd_java_metrics_index.html#non-commenting-source-statements-ncss
[5] https://github.com/AlDanial/cloc
[6] https://checkstyle.org/checks/sizes/methodlength.html
[7] https://pmd.github.io/
[8] https://docs.pmd-code.org/apidocs/pmd-java/7.0.0-rc4/net/sourceforge/pmd/lang/java/metrics/JavaMetrics.html#CYCLO

assign them to a risk as listed in table 2.8. Based on the LOC of the methods, GHD can calculate the relative risks compared to the total LOC and assign a SIG ranking as shown in table 2.9.

Duplication The tool GHD uses to detect duplicates of more than 6 lines is Simian Similarity Analyzer[9], a Java written tool with a lot of options such as lines threshold[10]. Simian goes through all files in the file list. With the relative percentage of LOC of all duplicates to the total LOC, GHD can determine the SIG ranking, as listed in table 2.10.

Comments By using regular expressions, GHD filters out the comment lines (single line, in line, and multi-line) from the source code in the files of the file list. GHD counts the number of lines in all returned comment blocks and classify the comments based on relevance (section 2.4). For classification, GHD uses a random forest classification model, which is implemented using the Scikit-learn [11] library. Based on the rankings in table 4.2 a ranking for comments is assigned.

With the collected data from the metrics, GHD can determine the final SIG maintainability scores for analysability, changeability, and stability. These scores are determined based on the average scores of the metrics, as described in table 5.1. All collected data will be stored in a database and is traceable by the UUID. Refer to table 5.9 for an overview of the collected data related to maintainability.

5.1.1. VALIDATION

To validate the calculations of the SIG maintainability model (section 2.3.4), we manually computed the metrics. We have picked a smaller repository for this task because it was not doable manually with larger ones. Our test repository was "Strassen-matrix-multiplication-Java"[12] (Strassen-matrix). If possible, we also computed the metrics manually for the "exchange-core" repository. When we executed our GHD tool, the calculated results of the SIG maintainability model for these repositories were as outlined in table 5.1.

VOLUME

The repository "Strassen-matrix" consists of only 1 Java file. This file contains 170 lines. According to our tool, the "Strassen.java" file has 110 lines of clean code. When we manually remove all the blank lines and comment blocks, we also end up with 110 lines. These are all lines of code, without comments. This includes imports, interfaces, class definitions and so on.

COMPLEXITY PER UNIT

We manually calculate the CC of several methods in the repositories "Strassen-matrix" and "exchange-core". We compare these calculations with the results from our tool, which uses the PMD tool [13]. The results are in table 5.2. We note that all methods in "Strassen-matrix" have a CC lower than 10, and as a result, they are all categorized at low risk (section 2.3.4). In Table 5.1, we can see that all lines (103) are classified at low risk. The number of lines is 103 because only the code of the methods is counted. The SIG complexity ranking is thus ++.

[9] https://simian.quandarypeak.com/
[10] https://simian.quandarypeak.com/docs/#options
[11] https://scikit-learn.org
[12] https://github.com/MoeidHeidari/Strassen-matrix-multiplication-Java
[13] https://pmd.github.io/

Metric	Strassen-matrix	exchange-core
Commit SHA	34f1861	2f85487
Total LOC (excluding comments or blank lines)	110	7538
Total lines (including comments, excluding blank lines)	139	9888
SIG volume ranking	++	++
LOC complexity per risk (only executable units)	low: 103 moderate: 0 high: 0 very high: 0	low: 4399 moderate: 307 high: 544 very high: 0
Percentage complexity per risk (only executable units)	low: 94% moderate: 0% high: 0% very high: 0%	low: 58% moderate: 4% high: 7% very high: 0%
LOC unit size per risk (only executable units)	low: 63 moderate: 40 high: 0 very high: 0	low: 3980 moderate: 396 high: 488 very high: 386
Percentage unit size per risk (only executable units)	low: 57% moderate: 36% high: 0% very high: 0%	low: 53% moderate: 5% high: 6% very high: 5%
SIG complexity ranking	++	*o*
SIG unit size ranking	−	+
Duplication line count	0	218
Duplication block count	0	50
Duplication percentage	0	3%
SIG duplication ranking	++	++
Comments ranking	+	++
SIG analysability ranking	+	++
SIG changeability ranking	++	+
SIG testability ranking	+	+

Table 5.1: SIG maintainabilty model results for the validation purpose

UNIT SIZE

We are manually counting the lines of code for multiple methods within the "Strassen-matrix" repository. The lines of code found by our tool, which uses PMD, do not match our manual count. PMD does not count the lines of code but the non commenting source statements (NCSS[14]). In essence, it is cleaner counting only ; and { as lines but does not match with our other metrics within our tool and disrupts other calculated metrics. We replaced PMD for counting lines with the CheckStyle[15] application, which follows the conventional loc counting method. This tool counts the loc for each method body (including { and }). In table 5.3 are the results of the validation count.

[14] `https://docs.pmd-code.org/apidocs/pmd-java/7.0.0-SNAPSHOT/net/sourceforge/pmd/lang/java/metrics/JavaMetrics.html#NCSS`

[15] `https://checkstyle.org/`

Repository	Commit SHA	File @ Method	CC	CC (Manual)
Strassen-matrix	34f1861	Strassen.java @ StrassenMultiply	2	2 (5 nodes, 5 edges)
Strassen-matrix	34f1861	Strassen.java @ main	5	5 (13 nodes, 16 edges)
Strassen-matrix	34f1861	Strassen.java @ addMatrices	3	3 (7 nodes, 8 edges)
Strassen-matrix	34f1861	Strassen.java @ subMatrices	3	3 (7 nodes, 8 edges)
Strassen-matrix	34f1861	Strassen.java @ printMatrix	3	3 (7 nodes, 8 edges)
Strassen-matrix	34f1861	Strassen.java @ divideArray	3	3 (7 nodes, 8 edges)
Strassen-matrix	34f1861	Strassen.java @ copySubArray	3	3 (7 nodes, 8 edges)
exchange-core	2f85487	ExchangeApi.java (298) @ publishBinaryData	3	3 (7 nodes, 8 edges)
exchange-core	2f85487	OrderBookDirectImpl.java (363) @ reduceOrder	6	6 (9 nodes, 13 edges)
exchange-core	2f85487	RiskEngine.java (521) @ canPlaceMarginOrder	5	5 (9 nodes, 12 edges)
exchange-core	2f85487	HashingUtils.java (68) @ checkStreamsEqual	4	4 (6 nodes, 8 edges)

Table 5.2: Cyclomatic Complexity Comparison

In the repository "Strassen-matrix" we can see that there is a single method with 40 LOC (moderate risk), and the rest all have a LOC that is no higher than 30 (low risk). In the result in table 5.1, under "LOC unit size per risk" we find values of 63 for low and 40 for moderate. So this measurement is valid.

Repository	Commit SHA	File @ Method	NCSS	LOC	LOC (manual)
Strassen-matrix	34f1861	Strassen.java @ StrassenMultiply	38	40	40
Strassen-matrix	34f1861	Strassen.java @ main	17	22	22
Strassen-matrix	34f1861	Strassen.java @ addMatrices	7	10	10
Strassen-matrix	34f1861	Strassen.java @ subMatrices	7	10	10
Strassen-matrix	34f1861	Strassen.java @ printMatrix	8	11	11
Strassen-matrix	34f1861	Strassen.java @ divideArray	4	5	5
Strassen-matrix	34f1861	Strassen.java @ copySubArray	4	5	5
exchange-core	2f85487	ExchangeApi.java (298) @ publishBinaryData	13	19	19
exchange-core	2f85487	SimpleEventsProcessor.java (38) @ sendTradeEvents	11	22	22
exchange-core	2f85487	SimpleEventsProcessor.java (67) @ sendTradeEvent	11	44	44
exchange-core	2f85487	L2MarketData.java (49) @ L2MarketData	9	10	10

Table 5.3: Unit size comparison

DUPLICATION

For measuring code duplicates, we use the Simian application[16]. We tested this application with the "exchange-core" repository and found 48 duplicates (blocks) of 6 or more lines. The total duplicated line count is 212 loc. Since Simian is one of the few tools that compares based on the number of lines, it is challenging to find a comparable tool. We can not use PMD for the comparison because it assesses based on a maximum number of tokens rather than lines. Because Simian is a widely used and also commercial tool, we consider the results to be valid.

COMPARISON WITH EXISTING SIG CALCULATIONS

At the Open University, assignments were created that make use of the SIG maintainability model to calculate metrics for several repositories. We imported the repositories: Jabberpoint[17], SmallSQL[18] and HyperSQL[19] into GHD and compared the results as shown in table 5.4. The rankings for analysability, changeability, and testability may differ because GHD does not use unit testing metric for the calculations, but uses the comments metric.

[16] `https://simian.quandarypeak.com/`
[17] `https://github.com/jpduits/jabberpoint`
[18] `https://github.com/jpduits/smallsql`
[19] `https://github.com/jpduits/hsqldb`

Metric	SmallSQL	HyperSQL	Jabberpoint
Total LOC (excluding comments or blank lines)	19346	143721	669
Total lines (including comments, excluding blank lines)	27411	211732	868
SIG volume ranking	++	+	++
LOC complexity per risk (only executable units)	low: 11375 moderate: 1653 high: 2510 very high: 1036	low: 74048 moderate: 20398 high: 17230 very high: 14574	low: 556 moderate: 0 high: 0 very high: 0
Percentage complexity per risk (only executable units)	low: 59% moderate: 9% high: 13% very high: 5%	low: 52% moderate: 14% high: 12% very high: 10%	low: 83% moderate: 0% high: 0% very high: 0%
LOC unit size per risk (only executable units)	low: 11461 moderate: 1138 high: 1882 very high: 2094	low: 63798 moderate: 11900 high: 16440 very high: 34112	low: 408 moderate: 72 high: 0 very high: 76
Percentage unit size per risk (only executable units)	low: 59% moderate: 6% high: 10% very high: 11%	low: 44% moderate: 8% high: 11% very high: 24%	low: 61% moderate: 11% high: 0% very high: 11%
SIG complexity ranking	−−	−−	++
SIG unit size ranking	*o*	−−	−
Duplication line count	646	9248	0
Duplication block count	158	1701	0
Duplication percentage	3%	6%	0%
SIG duplication ranking	+	*o*	++
SIG analysability ranking	+	*o*	+
SIG changeability ranking	*o*	*o*	++
SIG testability ranking	−	−−	+

Table 5.4: SIG maintainabilty model results for sample projects

COMMENTS

When calculating the metric to determine the percentage of relevant comments (section 2.4), we run a Python script that uses a Random Forest classifier. Because it is crucial to ensure the correctness of the classifications, we evaluate various formulas to measure precision, recall and accuracy. The formulas make use of the terms as listed in table 5.5. The formulas are defined as follows ([14]):

$$\text{Precision} = \frac{|TP|}{|TP| + |FP|} \text{ OR } \frac{|TN|}{|TN| + |FN|}$$

$$\text{Recall} = \frac{|TP|}{|TP| + |FN|} \text{ OR } \frac{|TN|}{|TN| + |FP|}$$

$$\text{Accuracy} = \frac{|TP| + |TN|}{|TP| + |TN| + |FP| + |FN|}$$

We have classified the comments for 5 repositories using our Python script, and then manually assigned the TP, FP, FN, or TN labels (see table 5.6). The spreadsheets we used for manual processing can be found in our GHD repository on GitHub [20]. Using these results, we calculated accuracy, precision, and recall. The results are listed in table 5.7. The definitions of relevant and auxiliary comments are described in section 2.4. We assess that accuracy, precision, and recall have very high values (with a maximum of 1), indicating that our classification script provides excellent and validated results.

Term	**Description**
True Positive (TP)	The model correctly predicts "RELEVANT"
False Positive (FP)	The model predicts "RELEVANT", but it is "AUXILIARY"
False Negative (FN)	The model predicts "AUXILIARY", but it is "RELEVANT"
True Negative (TN)	The model correctly predicts "AUXILIARY"

Table 5.5: Definitions of evaluation terms.

Repository	**Comments**	**Relevant**	**Auxiliary**	**FP**	**TP**	**FN**	**TN**
Strassen-matrix	16	11	5	0	11	5	0
cron-utils	445	360	85	0	360	1	84
exchange core	914	773	141	0	773	68	73
Zip4j	337	288	49	0	288	7	42
eureka	1669	1295	374	0	1295	16	358
Total	**3381**	**2727**	**654**	**0**	**2727**	**97**	**557**

Table 5.6: Classification evaluation

Repository	**Accuracy**	**Comments**	**Precision**	**Recall**
Strassen-matrix	0.69	Relevant comments Auxiliary comments	1 1	0.69 -
cron-utils	1	Relevant comments Auxiliary comments	1 0.99	0.99 1
exchange core	0,93	Relevant comments Auxiliary comments	1 0.52	0.92 1
Zip4j	0.98	Relevant comments Auxiliary comments	1 0.86	0.98 1
eureka	0.99	Relevant comments Auxiliary comments	1 0.96	0.99 1
Total	0.97	Relevant comments Auxiliary comments	1 0.85	0.97 1

Table 5.7: Accuracy, precision and recall of Random Forrest classifier

5.2. MEASURING THE COMMUNITY ASPECTS OF OSS PROJECTS

The community aspect of the quality model is measured using the GitHub metadata from our dataset, such as the Git commit history. Our GHD tool also stores a lot of data that is not currently used in this research but may be useful in the future. In the quality model,

[20] https://github.com/jpduits/ghd/tree/main/scripts/validation/comments

GHD measures the popularity of a repository using the metrics of Yamashita et al. [24] (section 2.2). To calculate these metrics, GHD filters Git metadata of a selected period from the dataset. GHD calculates the sticky and magnet value using 2 metrics.

Sticky To calculate the sticky value, GHD needs the contributors from the selected period and the contributors from the previous period. GHD retrieves this data using a MySQL query on the dataset. GHD queries all unique `author_id` of all commits within the selected period of the respective repository. It also includes the commits associated with pull requests that are (not yet) merged. GHD executes the same query for the previous period (determined based on the interval). GHD now has two sets of unique `author_id`. With these two sets of unique `author_id`, it can now determine which `author_id` have contributed to the repository in both the current and the previous period. GHD can now calculate the sticky value using the formula described in section 4.1.

Magnet The magnet value is determined based on the number of new contributors of the repository in the respective period compared to the total number of unique contributors. By using MySQL queries on the dataset, GHD can retrieve this information. It first queries all unique `author_id` of all commits, including the commits associated with pull requests, before selected period of the respective repository. Afterward, GHD does the same for the selected period only. It now has two sets of unique `author_id` allowing us to determine which contributors are new (have not appeared in any period before). It is now possible to calculate the magnet value using the formula as described in section 4.1.

With the calculated magnet and sticky values, GHD can determine the final OSS community category. All collected data from the metrics is stored in the database. Because our dataset contains a lot of metadata, GHD also stores a lot of unused information, such as the number of forks or stargazers in the selected period. Refer to table 5.9 for an overview of the collected data related to the community.

5.2.1. Validation

We can validate the metrics used for calculating Yamashita et al. OSS community categories (section 2.2) by performing several sample tests. For this sample test, we use the median values based on our measurement results in the database (sticky: 0.25, magnet: 0,0613385). We have performed calculations for "exchange-core" and "dubbo" [21]. We are measuring the period from 01-01-2019 to 02-07-2019. We first started with "exchange-core", the total number of developers until 02-07-2019 for this repository was 3. Of these, 1 was new during the measured period. It was noticeable that among these 3 developers, 2 shared the same email address but had different names. When we dug deeper, we discovered that our user table had quite a few duplicate users, all using the same email addresses. If we do not fix this, our measurements might not be right. To clear things up, we decided to tidy up the data in the database. We merged all users in the users table with the same email address and updated all references in linked tables. In this process, we picked the user with the longest name to represent them. Now, when we look at the same data, we see a total of 2 developers until 02-07-2019. This has an impact on the metrics of Yamashita et al.. This means that

[21] `https://github.com/apache/dubbo`

initially, the magnet value was:

$$\text{Magnet value} = \frac{1}{3} = 0.33$$

But now that the modification has been made, the result have changed:

$$\text{Magnet value} = \frac{1}{2} = 0.5$$

The sticky value also changed. We notice that before 01-01-2019, there was a single developer (according to the GitHub webpage). This was, according to our data, not the case before our merging process. For the sticky value, we look at the developers over the current period who were also active in the previous period, and the number of developers from the previous period. In this case, there are 2 developers in the current period, and 1 of them was also active in the previous period. The sticky value after our adjustments is calculated as:

$$\text{Sticky value} = \frac{1}{1} = 1$$

Because "exchange-core" is a repository with few developers, we have also conducted the same measurements with a larger repository "dubbo". This repository has the total number of 758 developers until 02-07-2019. Of these 758 developers, 156 developers were new in current period. There were 211 developers active in current period, of these, 46 developers were also active in the previous period. In the previous period, there were 264 developers active. When we translate this into the Yamashita et al. formulas, we get the following values:

$$\text{Magnet value} = \frac{156}{758} = 0.2058, \qquad \text{Sticky value} = \frac{46}{264} = 0.1742$$

According to the median thresholds we use, the magnet value is high and the sticky value is low. According to table 2.2, this repository falls into the fluctuating OSS community category in this period.
After running the GHD for the "exchange-core" and "dubbo" repositories, over the specified period, and after our database adjustments, we obtain the results as listed in table 5.8. The calculation results are correct and therefore valid.

Parameter	**exchange-core**	**dubbo**
Period Start Date	01-01-2019	01-01-2019
Interval in weeks	26	26
Period End Date	02-07-2019	02-07-2019
Previous Period Start Date	03-07-2018	03-07-2018
Previous Period End Date	01-01-2019	01-01-2019
Developers with Contributions (Previous Period)	1	264
Developers with Contributions (Current Period)	2	211
Developers with Contributions (Both Periods)	1	46
Developers (Current Period)	2	211
New Developers (Current Period)	1	156
Total Developers until and including current period	2	758
Sticky Value	1	0.1742
Magnet Value	0.5	0.2058
OSS community category	Attractive	Fluctuating

Table 5.8: Yamashita metric calculations for "exchange-core" and "dubbo"

5.3. COLLECTED DATA

When everything has been processed, our GHD tool returns the data for every repository from our dataset as shown in table 5.9. We can use this collected data for our research questions.

Table 5.9: Collected data

Related to	Name	Description
	`run_uuid`	UUID of run.
	`repository_id`	ID of the repository in the `project_states` table.
	`full_name`	Full name of repository on GitHub.
Community and maintainability	`period_start_date`	Start date of measurement period.
Community and maintainability	`period_end_date`	End date of measurement period.
Community	`previous_period_start_date`	Start date of previous period (for measuring Yamashita metrics).
Community	`previous_period_end_date`	End date of previous period (for measuring Yamashita metrics).
Community and maintainability	`interval_weeks`	Number of weeks for measurement period. Default value is 26 (6 months).
maintainability	`total_loc`	Total lines of code (excluding comments or blank lines)
maintainability	`total_kloc`	`total_loc` converted to thousands
maintainability	`total_lines`	Total lines of code (including comments, excluding blank lines). Used for comments metric (section 2.4).
maintainability	`volume_ranking`	Ranking for volume. Possible values: ++, +, *o*, − or −−.

maintainability	`loc_unit_size_per_risk`	Total LOC of methods, per risk (low, moderate, high of very high) based on `total_loc`. For example:
		
		`low: 14199 lines`
		`moderate: 1761 lines`
		`high: 971 lines`
		`very_high: 555 lines`

maintainability	`percentage_unit_size_per_risk`	Percentage LOC of methods, per risk (low, moderate, high of very high) based on `total_loc`. For example:
		
		`low: 58.768%`
		`moderate: 7.289%`
		`high: 4.019%`
		`very_high: 2.297%`

maintainability	`unit_size_ranking`	Ranking for unit size. Possible values: ++, +, *o*, − or −−.
maintainability	`loc_complexity_per_risk`	Total LOC of cyclomatic complexity per risk (low, moderate, high of very high) based on `total_loc`. For example:

The list continues on the next page

Table 5.9 – *Continuation from the previous page*

Related to	Name	Description

		`low: 32529 lines`
		`moderate: 3297 lines`
		`high: 945 lines`
		`very_high: 0 lines`

maintainability	`percentage_complexity_per_risk`	Percentage LOC of cyclomatic complexity per risk (low, moderate, high of very high) based on `total_loc`. For example:
		
		`low: 65.173%`
		`moderate: 6.606%`
		`high: 1.893%`
		`very_high: 0%`

maintainability	`complexity_ranking`	Ranking for complexity per unit. Possible values: ++, +, *o*, − or −−.
maintainability	`duplication_line_count`	Duplicated LOC with a threshold of 6 lines.
maintainability	`duplication_block_count`	Duplicated blocks of code with a threshold of 6 lines.
maintainability	`duplication_percentage`	Percentage duplicated LOC (`duplication_line_count`) of total LOC (`total_loc`).
maintainability	`duplication_ranking`	Ranking for duplication. Possible values: ++, +, *o*, − or −−.
maintainability	`comments_loc`	LOCO (lines of comments) of all comments.
maintainability	`comments_total`	Total number of comments (single line, multiple line or inline).
maintainability	`comments_relevant_percentage`	Percentage relevant comments.
maintainability	`comments_relevant`	Number of relevant comments.
maintainability	`comments_copyright`	Number of copyright comments.
maintainability	`comments_auxiliary`	Number of auxiliary comments.
maintainability	`comments_relevant_loc`	LOCO of relevant comments.
maintainability	`comments_copyright_loc`	LOCO of copyright comments.
maintainability	`comments_auxiliary_loc`	LOCO of auxiliary comments.
maintainability	`comments_percentage`	Percentage comments based on `total_lines`.
maintainability	`comments_ranking`	Ranking for comments. Possible values: ++, +, *o*, − or −−.
maintainability	`analysability_ranking`	Maintainability score for analysability. Possible values: ++, +, *o*, − or −−.
maintainability	`changeability_ranking`	Maintainability score for changeability. Possible values: ++, +, *o*, − or −−.
maintainability	`testability_ranking`	Maintainability score for testability. Possible values: ++, +, *o*, − or −−.

The list continues on the next page

Table 5.9 – *Continuation from the previous page*

Related to	**Name**	**Description**
maintainability	`overall_ranking`	Overall score for maintainability based on `analysability_ranking`, `changeability_ranking`, `testability_ranking`, and `stability_ranking`. For `stability_ranking`, a neutral ranking of *o* is used. Possible values: ++, +, *o*, − or −−.
community	`devs_with_contr_prev_period`	Developers with contributions in previous period.
community	`devs_with_contr_cur_period`	Developers with contributions in measurement period.
community	`devs_with_contr_prev_and_current_period`	Developers with contributions in current measurement period and previous period.
community	`sticky_value`	Calculated sticky value of Yamashita et al. metric [24].
community	`devs_cur_period`	Developers in current measurement period.
community	`devs_new_cur_period`	New developers in current measurement period.
community	`devs_total`	Total developers until current measurement period.
community	`magnet_value`	Calculated magnet value of Yamashita et al. metric [24].
community	`quadrant (OSS community category)`	Determined OSS community category of Yamashita et al. based on `sticky_value` and `magnet_value`. Possible values: attractive, fluctuating, stagnant and terminal.
community (future use)	`issues_count_current_period`	Number of issues measurement period.
community (future use)	`issues_count_total`	Total number of issues.
community (future use)	`stargazers_count_current_period`	Number of stargazers measurement period.
community (future use)	`stargazers_count_total`	Total number of stargazers.
community (future use)	`pull_requests_count_current_period`	Number of pull requests measurement period.
community (future use)	`pull_requests_count_total`	Total number of pull requests.
community (future use)	`forks_count_current_period`	Number of forks in measurement period.
community (future use)	`forks_count_total`	Total number of forks.
community (future use)	`bugs_current_period`	Number of bugs in measurement period.
community (future use)	`bugs_total`	Total number of bugs.
community (future use)	`bugs_closed_current_period`	Number of closed bugs in measurement period.
community (future use)	`bugs_closed_total`	Total number of closed bugs.

The list continues on the next page

Table 5.9 – *Continuation from the previous page*

Related to	**Name**	**Description**
community (future use)	`support_current_period`	Number of support issues in measurement period.
community (future use)	`support_total`	Total number of support issues.
community (future use)	`support_closed_current_period`	Number of closed support issues in measurement period.
community (future use)	`support_closed_total`	Total number of closed support issues.
community (future use)	`issues_average_duration_days`	Average days for closing issues.

6

SOFTWARE MAINTAINABILITY OF OSS COMMUNITY CATEGORIES

6.1. MEASURING

For our main research question: *how can software maintainability of OSS community categories as defined by Yamashita et al. [24] be characterized?* and sub question 3: *how does software maintainability change when open source software projects make transitions between OSS community categories?*, we need to measure the maintainability and community metrics for the 90 repositories from our dataset with GHD. For each repository, we measure different periods of 26 weeks, starting from 01-01-2019 to 28-06-2022. We use the measurement periods as described in table 6.1.

Start Date	End Date
01-01-2019	02-07-2019
02-07-2019	31-12-2019
31-12-2019	30-06-2020
30-06-2020	29-12-2020
29-12-2020	29-06-2021
29-06-2021	28-12-2021
28-12-2021	28-06-2022
28-06-2022	27-12-2022

Table 6.1: Measurement periods

There were four instances where a repository had no available commits for a period in 2019. We have excluded the data from those periods for that repository from our results. Therefore, for our 90 repositories, we end up with 716 measurements and 626 possible transitions. When everything has been executed, we have 8 periods of measurement results for each repository, with the last period ending on 27-12-2022. The thresholds for the OSS community categories are determined by the median of the sticky and magnet values of all repositories in all measured periods. The sticky threshold is 0.25 and the magnet threshold is 0.0613385 (section 2.2).

6.2. CHARACTERIZE MAINTAINABILITY OF OSS COMMUNITY CATEGORIES

The main research question (section 3.1.1), has to characterize the software maintainability of the OSS community categories. To characterize the software maintainability, we first take a single measurement result from each repository. We take the period 28-06-2022 to 27-12-2022 (26 weeks). We now have 90 measurement results over the same period.
To clarify our approach, we provide an explanation of 2 measurements. In our examples, we measure the repositories of square/retrofit and alibaba/nacos from the dataset. In the last commit of the measurement period we collected the developer related numbers as shown in table 6.2. As mentioned in section 2.2, we consider a developer as someone who committed code to Git for a project.

$$P_{i-1} = \text{28-12-2021 to 28-06-2022}$$

$$P_i = \text{28-06-2022 to 27-12-2022}$$

Description	**square/retrofit**	**alibaba/nacos**
Total active developers	424	713
New developers (did not appear in previous commits)	7	96
Developers current period P_i	9	137
Developers previous period $P_i - 1$	15	131
Developers with both periods P_i and $P_i - 1$	2	36

Table 6.2: Number of developers in specific period

If we translate this into the sticky and magnet values of Yamashita et al. we can determine the OSS community categories of these measurement period [24].

$$\text{Magnet value square/retrofit} = \frac{\text{New developers in } P_i = 7}{\text{Total developers} = 424} = 0.01650943$$

$$\text{Magnet value alibaba/nacos} = \frac{\text{New developers in } P_i = 96}{\text{Total developers} = 713} = 0,13464236$$

$$\text{Sticky value square/retrofit} = \frac{\text{Developers period } P_i\text{, with contributions in } P_{i-1} = 2}{\text{Developers period } P_{i-1} = 15} = 0.13333333$$

$$\text{Sticky value alibaba/nacos} = \frac{\text{Developers period } P_i\text{, with contributions in } P_{i-1} = 36}{\text{Developers period } P_{i-1} = 131} = 0,27480916$$

For square/retrofit both the magnet and the sticky value for period P_i are lower than the thresholds for the OSS community category. Therefore, this repository falls into the OSS community category terminal. This is the worst OSS community category and means that the respective repository had difficulty attracting and retaining developers during this period. The sticky and magnet values of alibaba/nacos for period P_i are both higher than the

thresholds and therefore fall into the best OSS community category attractive, which means that the project attracts and retains developers.
Now that we know the OSS community categories, we see in table 6.3 the results of maintainability metrics that were measured by GHD in the relevant period P_i.

Measurement	Ranking square/retrofit	Ranking alibaba/nacos
volume_ranking	++	+
complexity_ranking	−	−
unit_size_ranking	+	++
duplication_ranking	o	o
comments_ranking	+	o

Table 6.3: Measurements and rankings for square/retrofit and alibaba/nacos in period P_i (28-06-2022 to 27-12-2022)

If we map the rankings from table 6.3 back to the maintainability characteristics, we obtain the values as shown in table 6.4. See section 2.3.4 to understand how the maintainability characteristics are determined.

Maintainability characteristic	Ranking square/retrofit	Ranking alibaba/nacos
analyzability_ranking	+	+
changeability_ranking	o	o
testability_ranking	o	o
overall_ranking	o	o

Table 6.4: Maintainability characteristics for square/retrofit in period P_i (28-06-2022 to 27-12-2022)

OSS community category	Count
Attractive (high sticky, high magnet)	16
Fluctuating (low sticky, high magnet)	12
Stagnant (high sticky, low magnet)	31
Terminal (low sticky, low magnet)	31
Total	90

Table 6.5: OSS community categories in period P_i (28-06-2022 to 27-12-2022)

We perform these same measurements for all 88 other repositories over the same period. For a complete overview of all measurement results, download the spreadsheet on GitHub[1]. The total of 90 repositories can be divided into the OSS community categories as shown in table 6.5. The bar chart of figure 6.1, shows the rankings of the maintainability characteristics per OSS community category for period P_i.
In the scatter plots of figures 6.2a, 6.2b, 6.2c, 6.2d we visualize maintainability characteristics of analysability, changeability and testability, respectively. On the horizontal axis, the sticky value is displayed, and the vertical axis shows the magnet value. The cross in the heat map is drawn at the median of the sticky and magnet values from our measurements. This provides insight into which OSS community category a repository falls into. The colored circles represent the repositories, with the color representing the ranking of the quality of the respective maintainability characteristic.

[1] https://github.com/jpduits/ghd_results/blob/main/results.ods

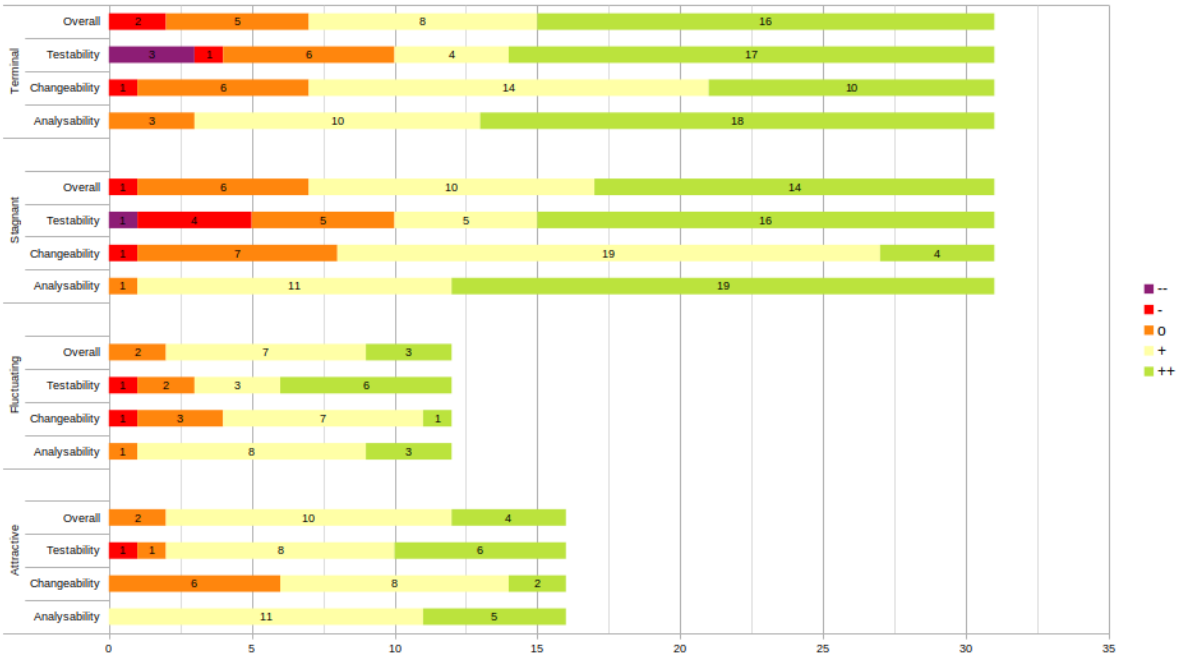
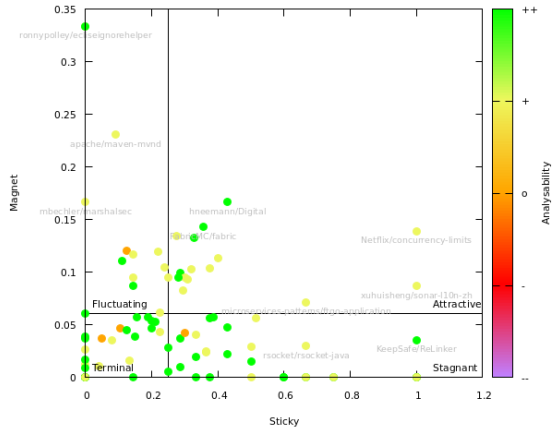
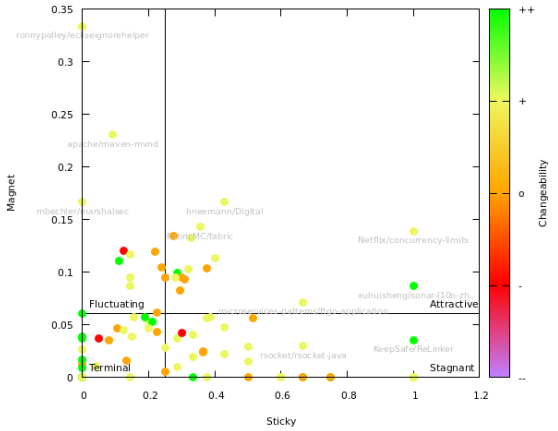


Figure 6.1: Maintainability characteristics per OSS community category in period P_i (28-06-2022 to 27-12-2022)

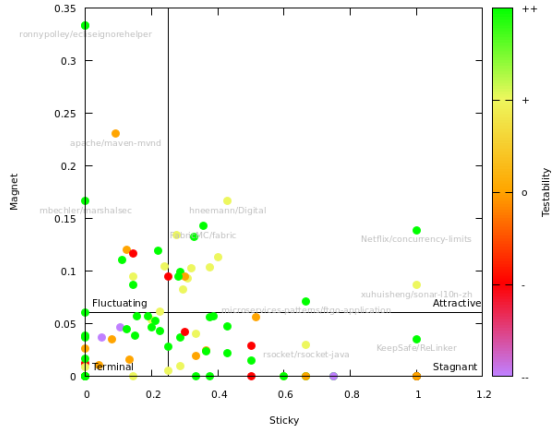
The scatter plots only provide insight for a specific period. As shown in table 6.6, we have measured our repositories in 8 different periods in our dataset. To visualize these periods, we have created custom heatmaps that displays each maintainability characteristic ranking for each period. For each ranking, we have created 2 heatmaps. The left heatmap shows the OSS community categories as columns and the periods as rows. For each OSS community category of the respective period, a repository is indicated by a block, with the color indicating the ranking. The results from our example, see table 6.5, are shown on the bottom period row of the heatmap. The right heatmap shows the same data in percentages. A block is 100%, which is filled with the percentages of the characteristic rankings.



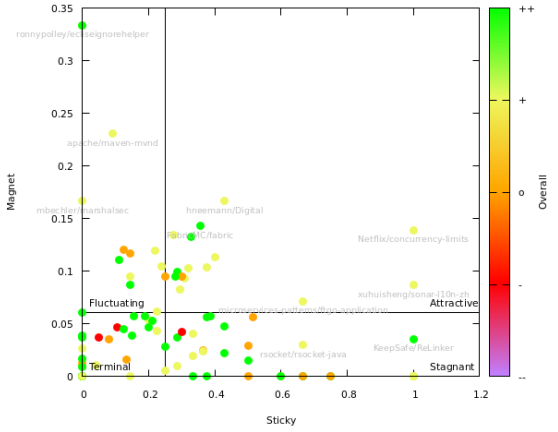
(a) OSS community categories and analysability in period P_i



(b) OSS community categories and changeability in period P_i



(c) OSS community community categories and testability in period P_i



(d) OSS community categories and overall maintainability ranking in period P_i

Figure 6.2: The maintainability rankings and OSS community categories for period P_i (28-06-2022 to 27-12-2022)

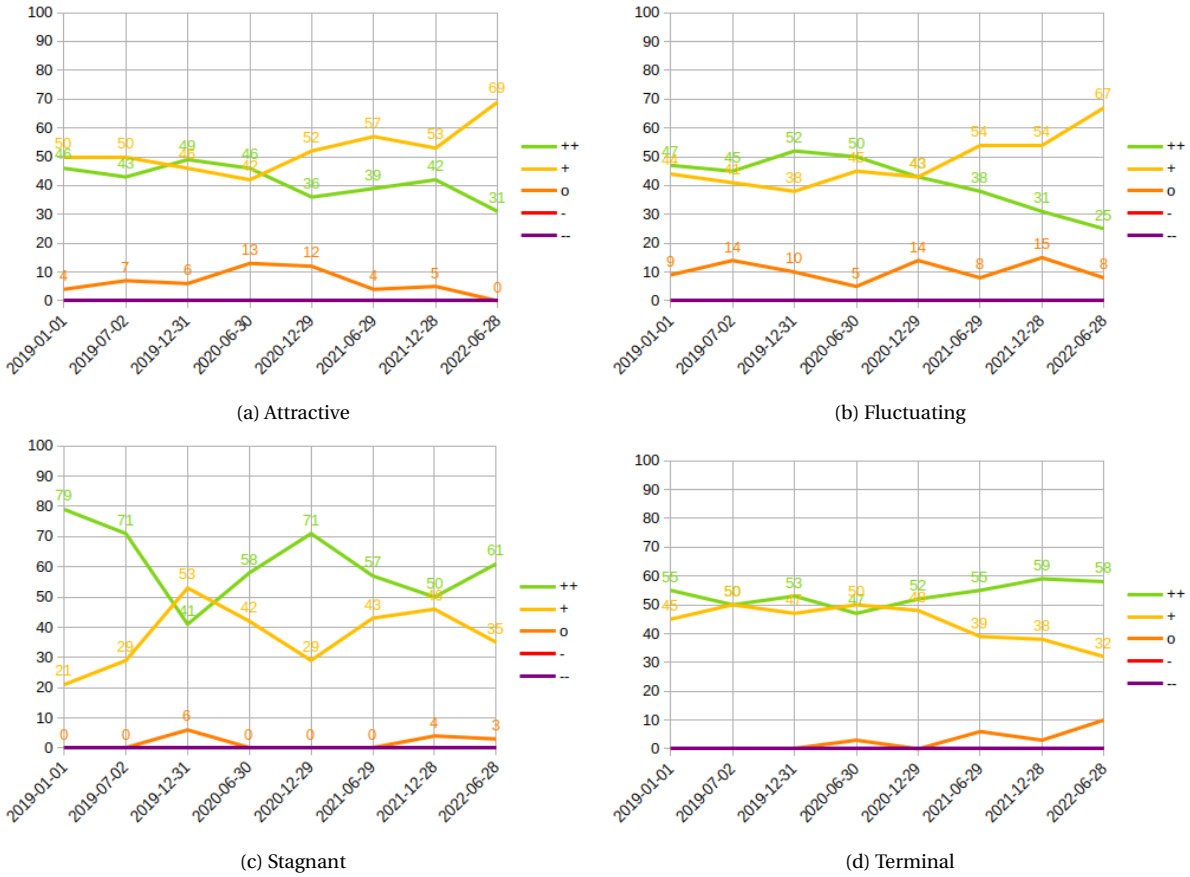
OSS community category	**Period 1** 01-01-2019 02-07-2019		**Period 2** 02-07-2019 31-12-2019		**Period 3** 31-12-2019 30-06-2020		**Period 4** 30-06-2020 29-12-2020		**Period 5** 29-12-2020 29-06-2021		**Period 6** 29-06-2021 28-12-2021		**Period 7** 28-12-2021 28-06-2022		**Period 8** 28-06-2022 27-12-2022	
Attractive	28	32.18%	30	33.71%	35	38.89%	24	26.67%	25	27.78%	23	25.56%	19	21.11%	16	17.78%
Fluctuating	34	39.08%	22	24.72%	21	23.33%	22	24.44%	21	23.33%	13	14.44%	13	14.44%	12	13.33%
Stagnant	14	16.09%	17	19.10%	17	18.89%	12	13.33%	21	23.33%	23	25.56%	26	28.89%	31	34.44%
Terminal	11	12.64%	20	22.47%	17	18.89%	32	35.56%	23	25.56%	31	34.44%	32	35.56%	31	34.44%
Total	87		89		90		90		90		90		90		90	

Table 6.6: OSS community categories per period

(a) Number of analysability rankings per period

(b) Percentages of the analysability rankings per period

Figure 6.3: Heatmaps of analysability ranking per OSS community category



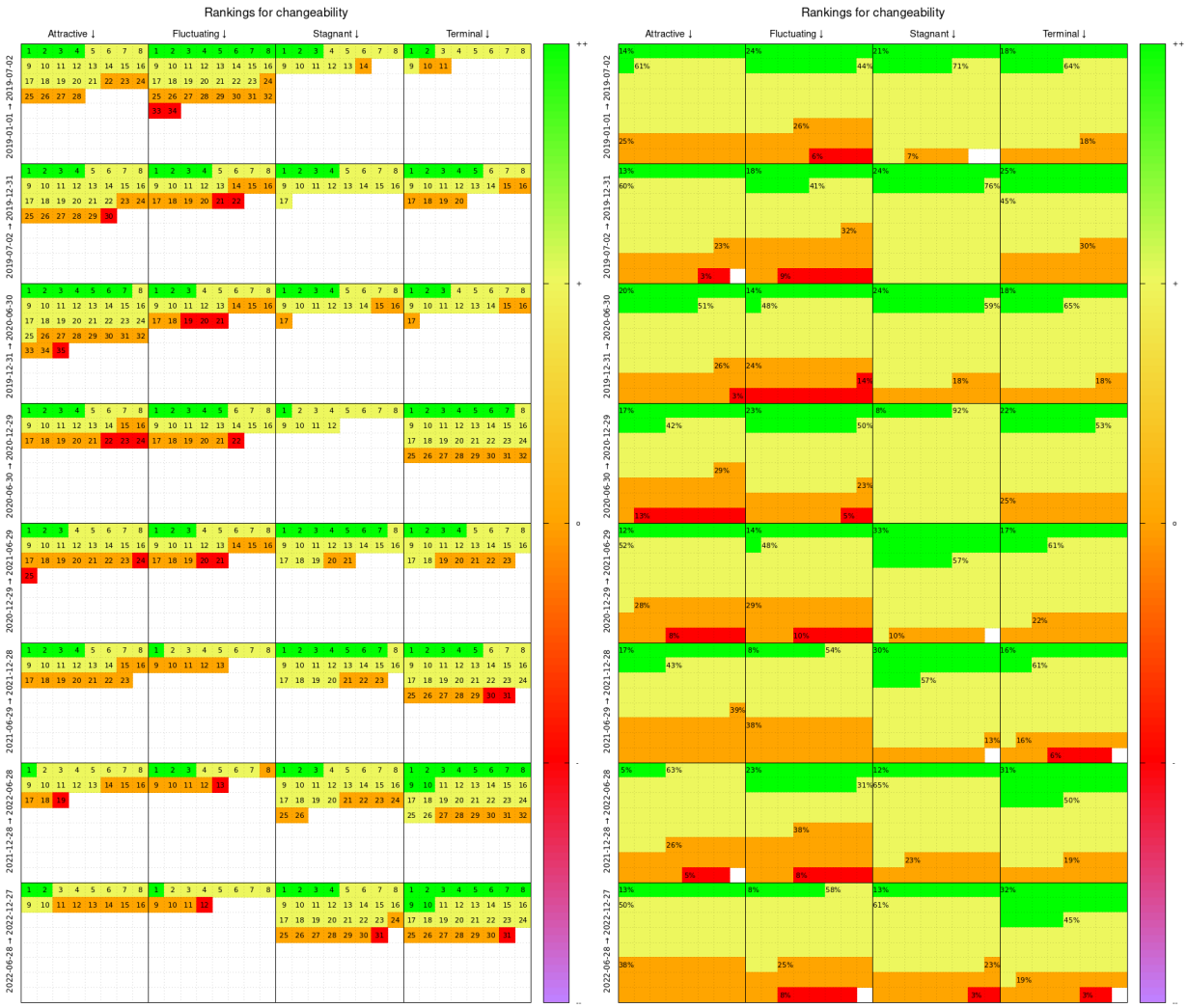
(a) Attractive

(b) Fluctuating

(c) Stagnant

(d) Terminal

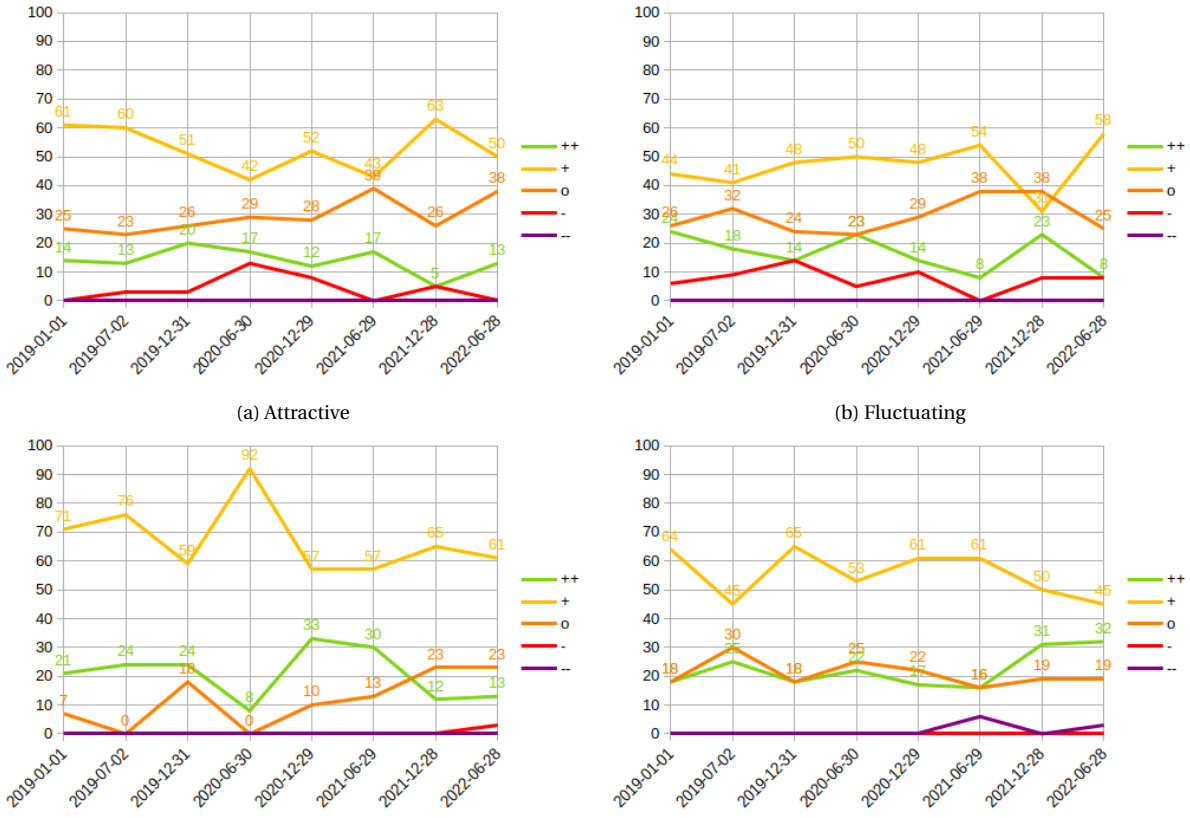
Figure 6.4: Line chart of analysability per period in percentages



(a) Number of changeability rankings per period

(b) Percentages of the changeability rankings per period

Figure 6.5: Heatmaps of changeability ranking per OSS community category



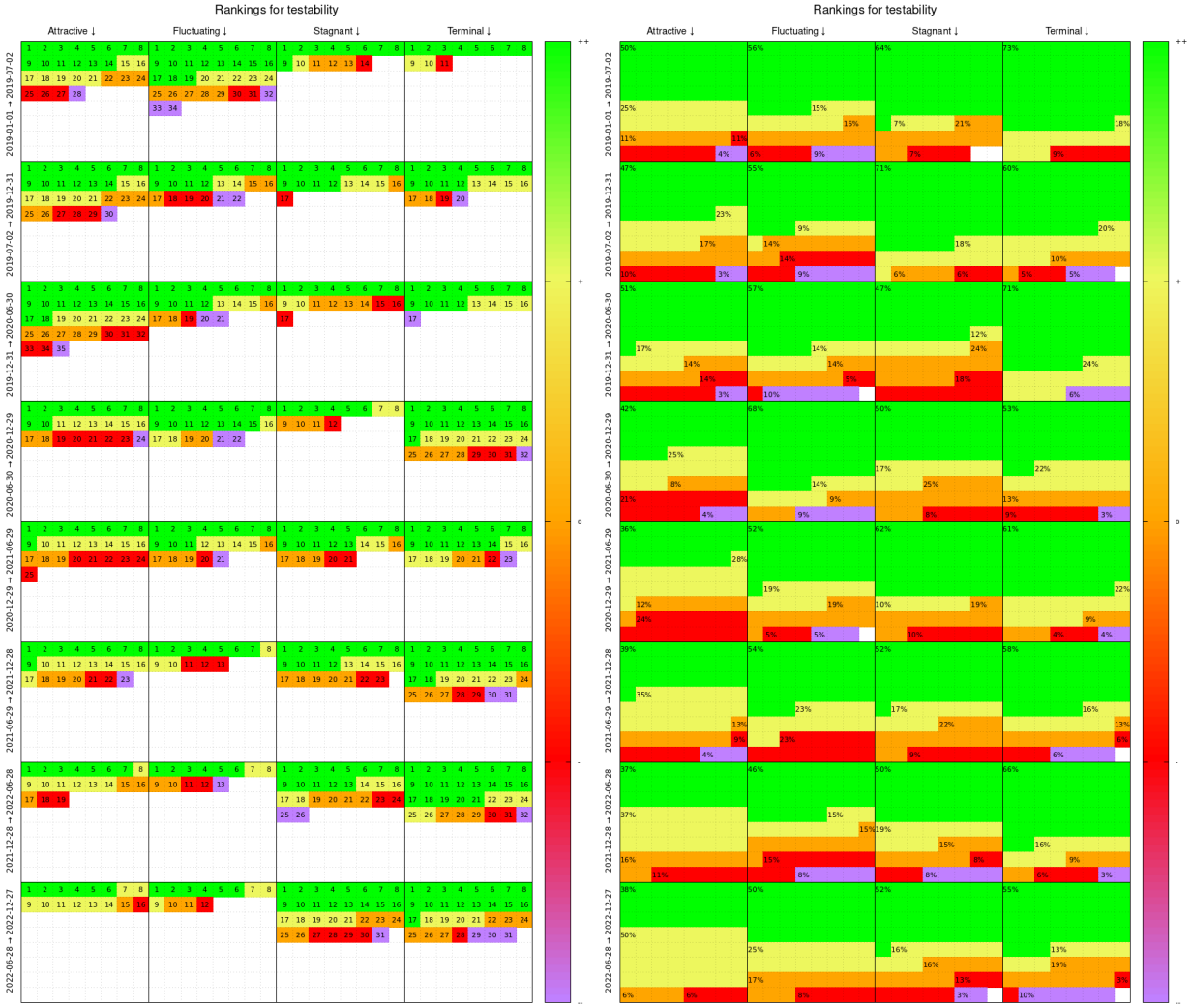
(a) Attractive

(b) Fluctuating

(c) Stagnant

(d) Terminal

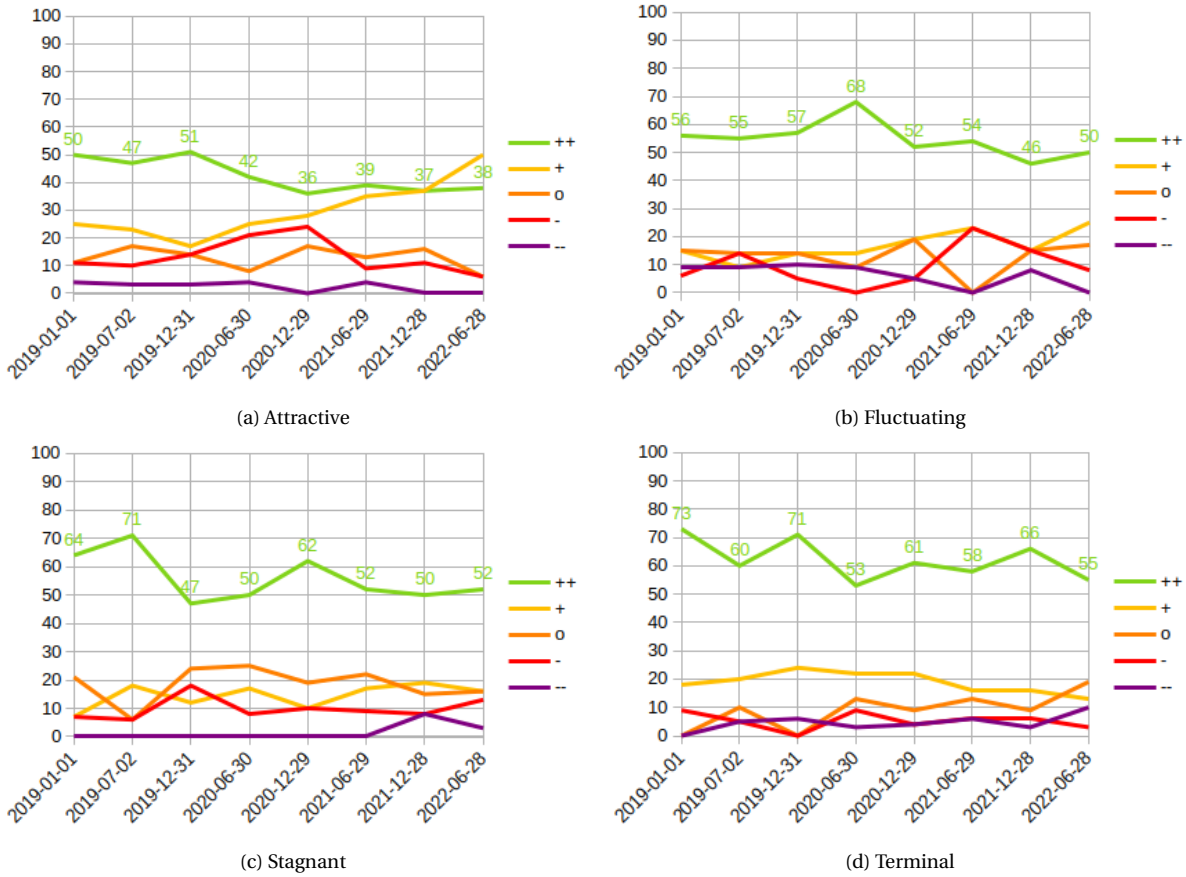
Figure 6.6: Line chart of changeability per period in percentages



(a) Number of testability rankings per period

(b) Percentages of the testability rankings per period

Figure 6.7: Heatmaps of testability ranking per OSS community category



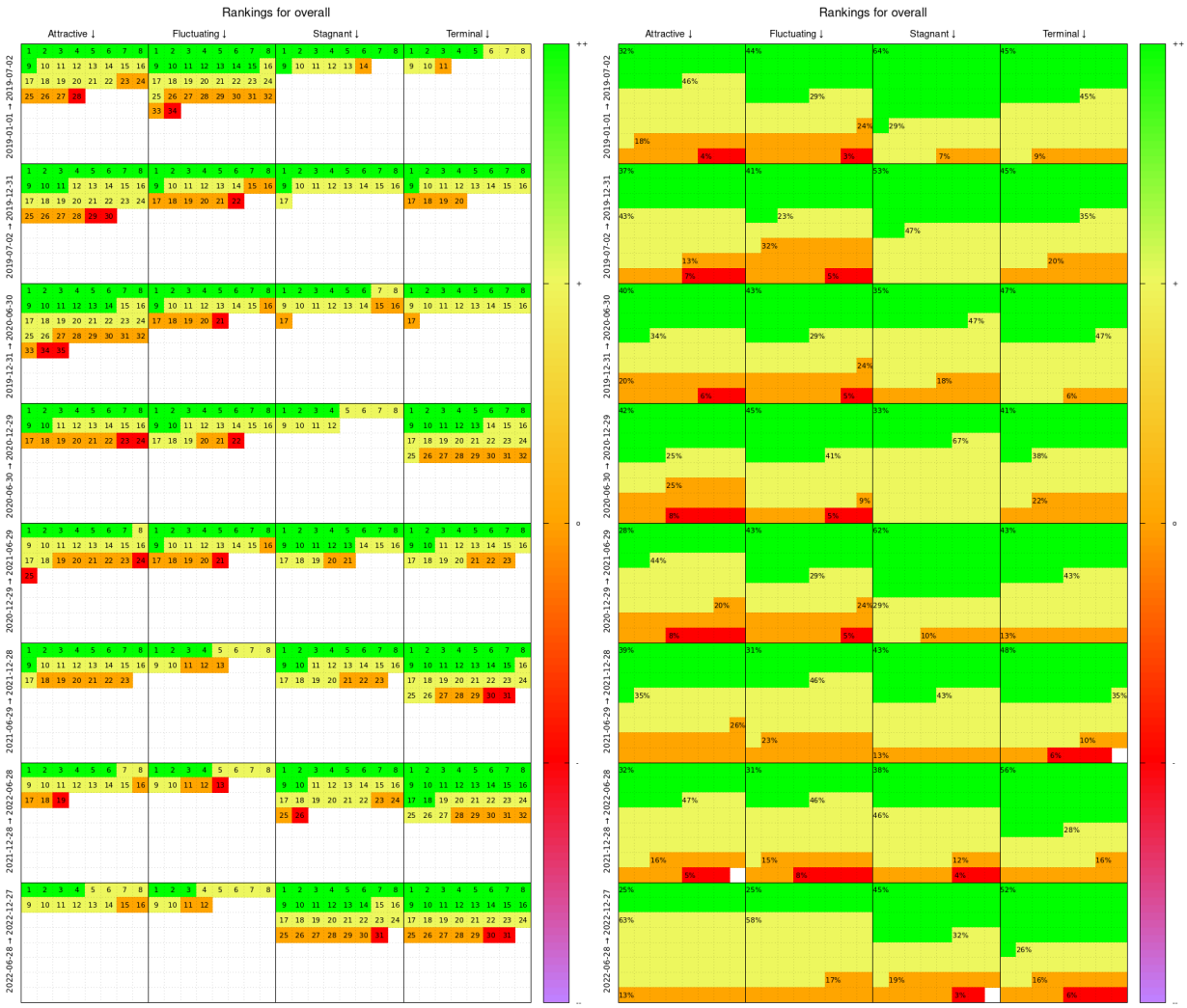
(a) Attractive

(b) Fluctuating

(c) Stagnant

(d) Terminal

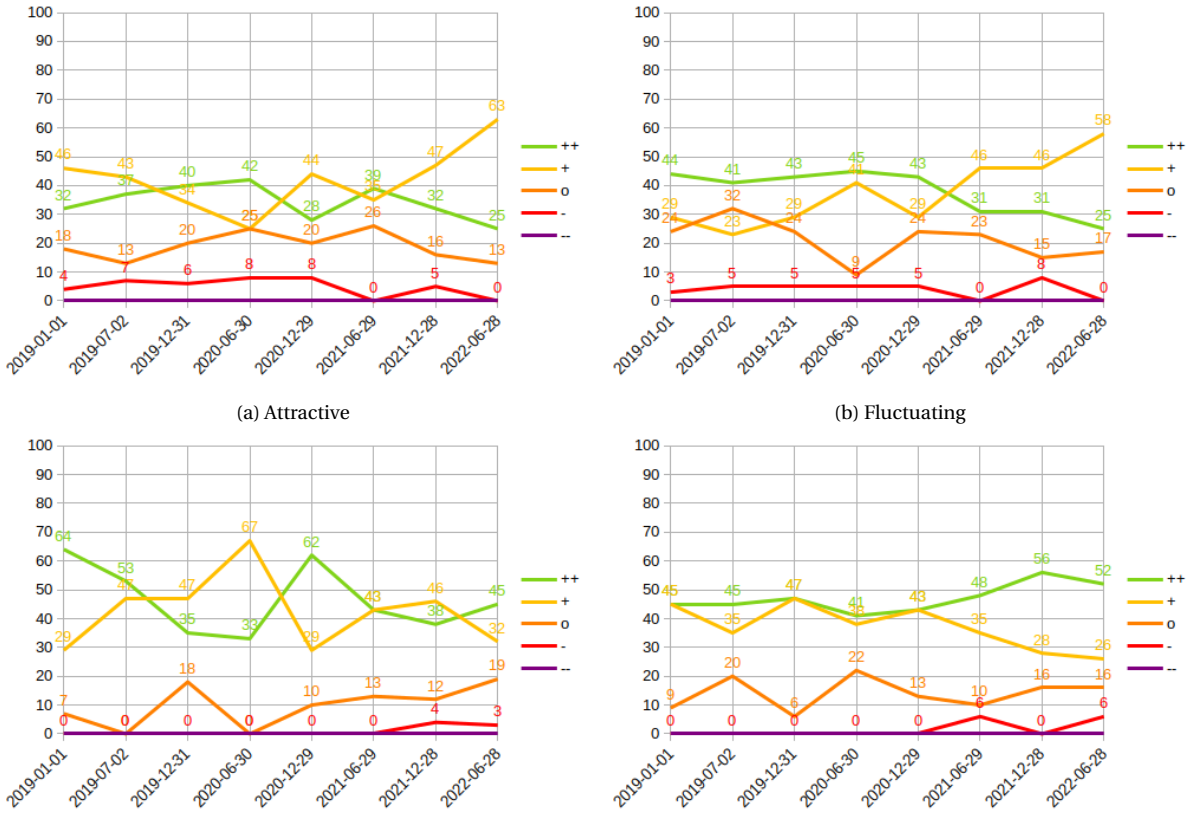
Figure 6.8: Line chart of testability per period in percentages



(a) Number of overall rankings per period

(b) Percentages of the overall rankings per period

Figure 6.9: Heatmaps of overall maintainability ranking per OSS community category



(a) Attractive

(b) Fluctuating

(c) Stagnant

(d) Terminal

Figure 6.10: Line chart of overall maintainability per period in percentages

6.3. THE MAINTAINABILITY OF OSS COMMUNITY CATEGORIES AFTER TRANSITIONS

To answer sub question 3: *how does software maintainability change when OSS projects make transitions between OSS community categories* (section 3.1.4), we determine for all our measurements from section 6.1 whether the OSS community category has changed or remained the same. When a project changes from a OSS community category, for example from attractive to stagnant, we call this a transition. When a project does not change OSS community categories, no transition occurs. For each repository, we have 8 measurement periods, so there are 7 possible periods for a transition between OSS community categories. See figure 6.11 for a visualization of the transitions periods. In total we have 90 repositories with 716 measurement periods and 626 possible transitions. Table 6.7 shows.

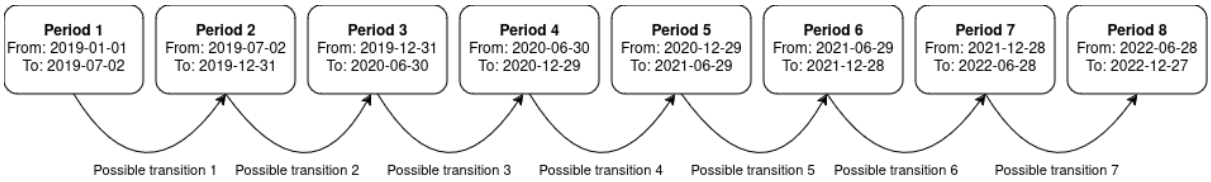


Figure 6.11: Transitions between periods for a repository

OSS community category	Measurement count
Attractive	200
Fluctuating	158
Stagnant	161
Terminal	197
Total measurement periods	716

Table 6.7: Measured OSS community categories in all 716 periods

For each transition, we analyzed whether the maintainability characteristics ranking have increased (↑), decreased (↓), or remained the same compared to the previous period (=). Table 6.8 provides an overview of the analysis. We applied the same principle to the metrics used, the results are in table 6.9. Figure 6.12a shows the transitions and the associated changing overall maintainability characteristics. The periods in which no transition occurred and the associated changing overall maintainability characteristics are shown in figure 6.12b.

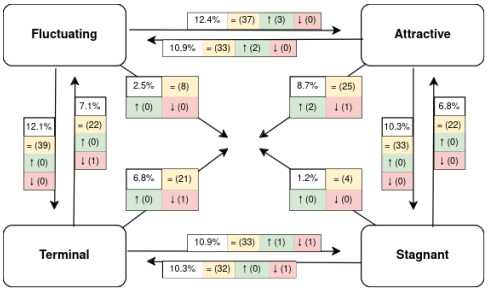
6.3.1. CONCLUSION - ANSWERING SUB QUESTION 3

In our measured periods, there were 626 possible transitions, of which actual transitions to another OSS community category occurred in 322 periods (51%). In 304 periods, no transition occurred and the OSS community category remained the same. When we analyze tables 6.8 and 6.9, it is noticeable that after transitions, a small percentage of changes can be observed in both the maintainability characteristics and the source code properties. Looking at the heatmaps of 6.2, you might expect that some changes in the maintainability characteristics would occur in every period, but this is not the case. When the OSS community category remains attractive, there are generally some more changes visible, but even here the percentages are low. The changes primarily occur in the transitions between OSS community categories but not in maintainability characteristics. When we combine all 4 OSS

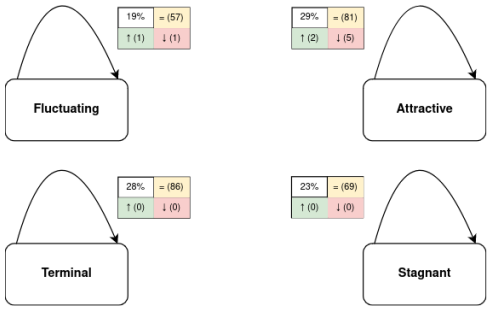
Transition	Count	%	Analysability			Changeability			Testability			Overall		
			=	↑	↓	=	↑	↓	=	↑	↓	=	↑	↓
Stagnant → Terminal	33	10.25%	32	0	1	32	0	1	33	0	0	32	0	1
Stagnant → Fluctuating	4	1.24%	4	0	0	4	0	0	4	0	0	4	0	0
Stagnant → Attractive	22	6.83%	22	0	0	22	0	0	20	0	2	22	0	0
Terminal → Stagnant	35	10.87%	33	1	1	33	1	1	34	0	1	33	1	1
Terminal → Fluctuating	23	7.14%	21	0	2	21	0	2	21	0	2	22	0	1
Terminal → Attractive	22	6.83%	21	0	1	21	1	0	20	1	1	21	0	1
Fluctuating → Stagnant	8	2.48%	8	0	0	8	0	0	8	0	0	8	0	0
Fluctuating → Terminal	39	12.11%	39	0	0	38	0	1	39	0	0	39	0	0
Fluctuating → Attractive	40	12.42%	38	2	0	36	2	2	36	3	1	37	3	0
Attractive → Stagnant	33	10.25%	33	0	0	33	0	0	33	0	0	33	0	0
Attractive → Terminal	28	8.70%	26	2	0	28	0	0	27	0	1	25	2	1
Attractive → Fluctuating	35	10.87%	33	1	1	34	0	1	31	3	1	33	2	0
Transitions total	322	100%	310	6	6	310	4	8	306	7	9	309	8	5

No transition	Count	%	Analysability			Changeability			Testability			Overall		
			=	↑	↓	=	↑	↓	=	↑	↓	=	↑	↓
Stagnant → Stagnant	71	23.36%	69	1	1	68	0	3	66	1	2	67	0	2
Terminal → Terminal	86	28.29%	86	0	0	85	1	0	85	1	0	86	0	0
Fluctuating → Fluctuating	59	19.41%	59	0	0	56	2	1	57	1	1	57	1	1
Attractive → Attractive	88	28.95%	79	3	6	79	6	3	85	1	2	81	2	5
No transition total	304	100%	293	4	7	288	9	7	293	4	5	291	3	8
Total	626													

Table 6.8: Transitions between OSS community categories and their changing maintainability characteristics



(a) Transitions between OSS community categories and their associated changing overall maintainability characteristics



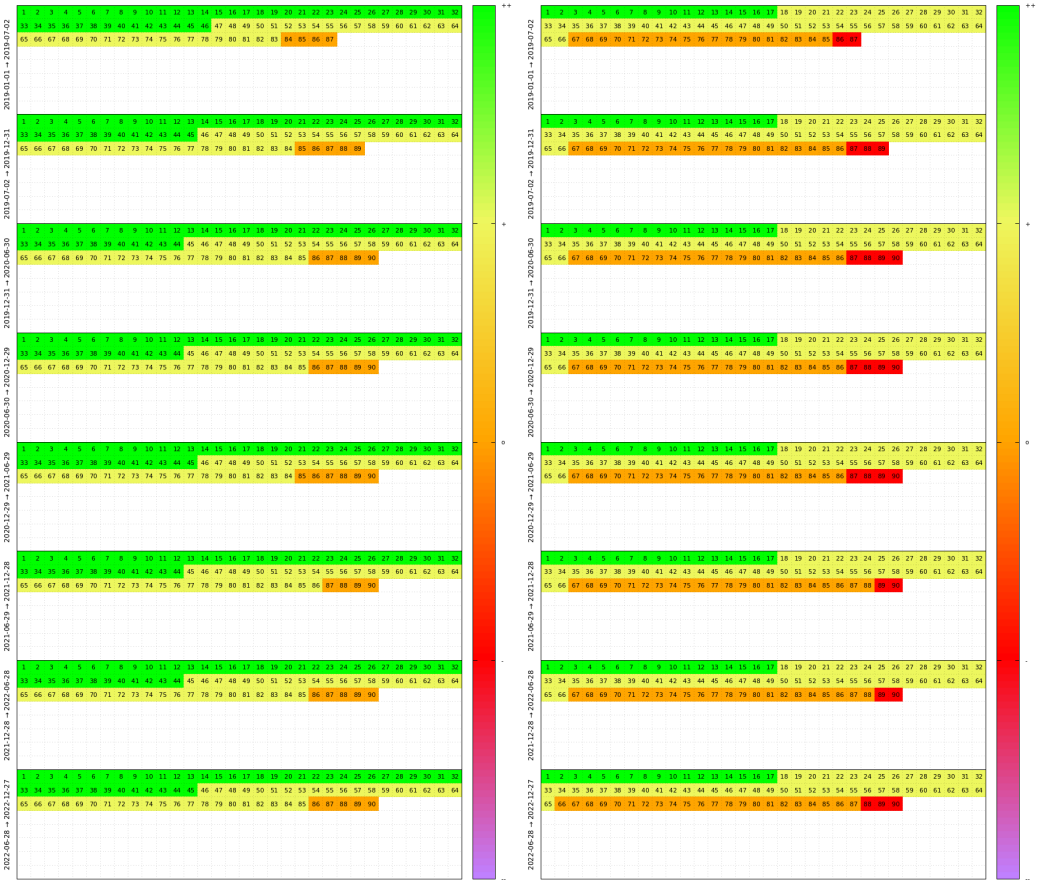
(b) No transitions between OSS community categories and the associated changing overall maintainability characteristics

Figure 6.12

community categories, we get a different overview as shown in figure 6.13. Now we see that the maintainability characteristics remain relatively stable in each period, with only some minor changes. The conclusion we can draw from this data is that a transition has minimal effect on the maintainability characteristics.

6.4. CONCLUSION - ANSWERING MAIN RESEARCH QUESTION

Looking at table 6.6, we start with the attractive (28 repositories, 32%) and fluctuating (34 repositories, 39%) OSS community categories having the highest percentages in the first period (01-01-2019). But these percentages slowly go down through the periods, ending up at 18% (16 repositories) for attractive and 13% (12 repositories) for fluctuating in the last period (28-06-2022). The stagnant and terminal OSS community categories start with lower percentages in the first period, 16% (14 repositories) and 13% (11 repositories) respectively. But by the last period both categories increasing to 34% (31 repositories). The progression of these OSS community categories is shown in the line chart of figure 6.14. Based on these findings from our dataset, we can conclude that there is a reasonable chance that an OSS



(a) Analysability

(b) Changeability

(c) Testability

(d) Overall

Figure 6.13: Total rankings of maintainability characteristics

Transition	Count	%	Complexity			Volume			Unit size			Duplication			Comments		
			=	↑	↓	=	↑	↓	=	↑	↓	=	↑	↓	=	↑	↓
Stagnant → Terminal	33	10.25%	33	0	0	33	0	0	33	0	0	32	0	1	33	0	0
Stagnant → Fluctuating	4	1.24%	4	0	0	4	0	0	4	0	0	4	0	0	4	0	0
Stagnant → Attractive	22	6.83%	21	0	1	22	0	0	21	0	1	19	0	3	22	0	0
Terminal → Stagnant	35	10.87%	33	1	1	35	0	0	34	0	1	33	1	1	34	1	0
Terminal → Fluctuating	23	7.14%	22	0	1	23	0	0	21	0	2	22	0	1	21	1	1
Terminal → Attractive	22	6.83%	21	1	0	22	0	0	20	1	1	20	0	2	22	0	0
Fluctuating → Stagnant	8	2.48%	8	0	0	8	0	0	8	0	0	8	0	0	8	0	0
Fluctuating → Terminal	39	12.11%	38	0	1	39	0	0	38	0	1	39	0	0	39	0	0
Fluctuating → Attractive	40	12.42%	38	1	1	40	0	0	38	2	0	34	3	3	37	2	1
Attractive → Stagnant	33	10.25%	33	0	0	33	0	0	33	0	0	32	1	0	33	0	0
Attractive → Terminal	28	8.7%	27	0	1	28	0	0	27	1	0	27	1	0	27	1	0
Attractive → Fluctuating	35	10.87%	33	1	1	34	0	1	33	2	0	33	1	1	35	0	0
Transitions total	322	100%	311	4	7	321	0	1	310	6	6	303	7	12	315	5	2
No transition	**Count**	**%**	**Complexity**			**Volume**			**Unit size**			**Duplication**			**Comments**		
			=	↑	↓	=	↑	↓	=	↑	↓	=	↑	↓	=	↑	↓
Stagnant → Stagnant	71	23.36%	67	0	4	70	0	1	66	3	2	69	1	1	71	0	0
Terminal → Terminal	86	28.29%	84	2	0	86	0	0	85	0	1	85	0	1	86	0	0
Fluctuating → Fluctuating	59	19.41%	58	0	1	59	0	0	57	1	1	57	1	1	55	2	2
Attractive → Attractive	88	28.95%	85	1	2	86	0	2	78	3	7	79	4	5	82	4	2
No transition total	304	100%	294	3	7	301	0	3	286	7	11	290	6	8	294	6	4
Total	626																

Table 6.9: Transitions between OSS community categories and their source code properties

repository will decrease in popularity and struggle to attract new developers. This conclusion seems to align with the fact that after a while, OSS projects may enter a more stabilized phase, which is not attractive or dynamic enough to attract new developers, and mainly rely on their core developers.
With the visualizations we have created (heatmaps and line charts), we can overview the maintainability characteristics per OSS community category and assess if there is a relationship.

Analysability The ranking percentages for analysability are good for all OSS community categories and remain good over time. It is noticeable that the stagnant and terminal OSS community categories significantly more often score a ++ ranking. At the attractive and fluctuating OSS community categories the ++ rankings slowly change to + rankings. Based on figure 6.4, we can conclude that a repository in a stagnant OSS community category is likely to have the best analysability characteristics for maintainability.

Changeability For all the OSS community categories, the changeability most often ranks at +, with stagnant and terminal having the higher percentages. Attractive and fluctuating OSS community categories more often have a − ranking, than the stagnant and terminal OSS community categories. But also, the percentage of *o* rankings is higher than the percentage of ++ rankings. It is noticeable that for the terminal OSS community category, in the last 2 periods, the percentage ++ rankings, significantly have increased. Based on this analysis and the line charts in figure 6.6, we can conclude that the stagnant OSS community category has likely to best changeability characteristics for maintainability.

Testability In the case of testability, a variety of ranking percentages are observed. However, across all OSS community categories, the testability characteristics are mostly acceptable (++ or +). Interestingly, the percentage of ++ rankings is lower for the

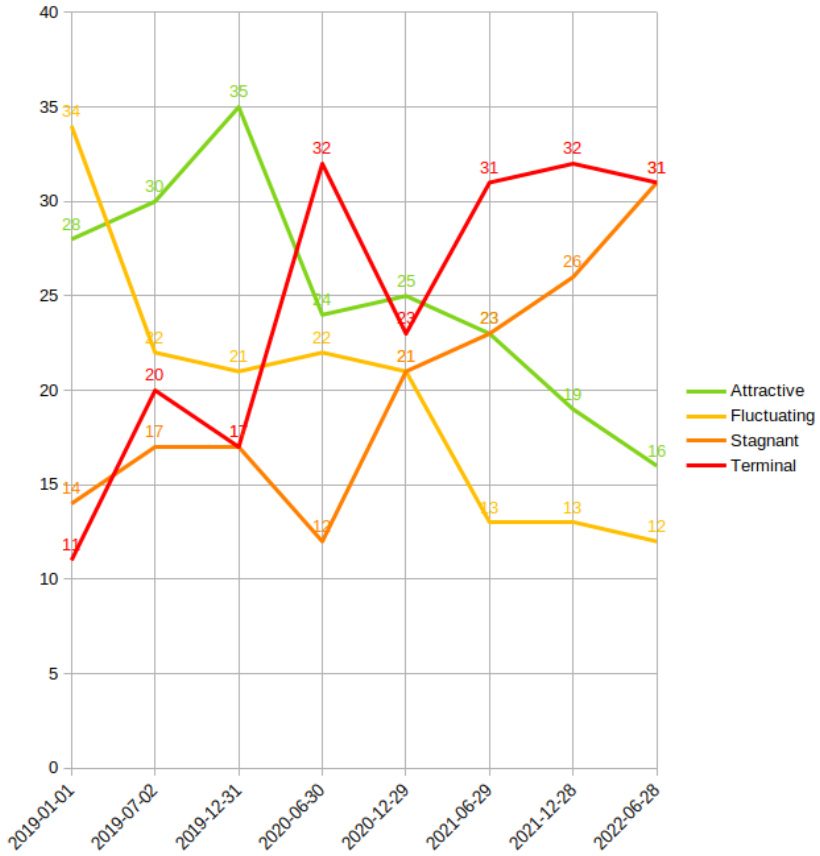


Figure 6.14: Line chart for number of repositories per OSS community categories and period

attractive OSS community category and highest for the terminal OSS community category. Furthermore, the terminal OSS community category has the lowest percentage of − and −− rankings. From this, we can conclude that terminal projects often have better testability characteristics for maintainability.

Overall When we look at the percentages of the overall rankings, we see that the average maintainability is good enough. The total percentages for the better rankings (++ and +) are higher for terminal and stagnant.

The final conclusion we can draw for our main research question (section 3.1.1) is that if it becomes difficult for a repository to attract and retain new developers over time, which is often the case, it is not necessarily detrimental to the maintainability characteristics. As shown in figure 6.10, the results seem to indicate that projects with a small and stable developer base score better on maintainability than popular repositories with a high turnover of new developers.

6.5. A MODEL TO PREDICT THE MAINTAINABILITY FOR AN OSS COMMUNITY CATEGORY

To answer sub-question 4: *How can we propose a predicting model for OSS software maintainability?* (section 3.1.5), it was necessary to create an additional tool that uses a regression based model. This tool should make it possible to specify an OSS community category, with 3 corresponding maintainability rankings, and based on this information, predict the

maintainability of the 4 possible future transitions. To build this tool, the Scikit-learn[2] library was used. The tool consists of 3 parts: an adjusted dataset, a training script, and a prediction script. The source code for this tool is available on GitHub[3].

6.5.1. Dataset

To efficiently train our model, we have added four columns to our dataset that we can use as features. For each record, the rankings and the OSS community category of the previous period are added, if possible. With these added rankings, the model can be trained. 80% of the dataset is used for training, 20% is used as the test set.

6.5.2. Train a linear regression model

We have built a script that trains a linear regression model. For each period, we use the previous maintainability rankings and the current OSS community category as input variables (features) and the current rankings as output (targets). We use these variables with the standard functionality to train a linear regression model from the Scikit-learn library. After training, the model is tested (see validation) and stored. The model can now be used in the prediction script.

6.5.3. Prediction

The prediction script predicts future maintainability rankings for each possible OSS community category. It asks the trained model to make a prediction based on the given features. These features are described in table 6.10.

Feature	Description
oss_category	The future OSS community category to predict (attractive, stagnant, fluctuating, or terminal).
prev_oss_category	The current OSS community category (in the future, this is the previous OSS community category). This value is entered as an argument of the script.
prev_rankings	The current maintainability rankings (analysability, changeability, testability). These values are entered as an arguments of the script.

Table 6.10: Features used for the prediction model

When the described features are provided to the model, we receive the predicted maintainability rankings back. To get predictions for all 4 OSS community categories, the model is consulted 4 times. The received rankings are translated into the SIG rankings and displayed on the screen per OSS community category. See listing 6.1 for an example of the executed script.

```
$ python predict.py Stagnant 3 3 4

Current OSS community Category: Stagnant
```

[2] https://scikit-learn.org
[3] https://github.com/jpduits/maintainability_predict

```
Current rankings:
Analysability:  o       (3.00)
Changeability:  o       (3.00)
Testability:    +       (4.00)

==============================

If next period is Attractive,
the predicted rankings are:

Analysability:  o       (3.09)
Changeability:  o       (2.96)
Testability:    +       (3.89)

==============================

If next period is Fluctuating,
the predicted rankings are:

Analysability:  o       (3.08)
Changeability:  o       (2.93)
Testability:    +       (3.89)

==============================

If next period is Stagnant,
the predicted rankings are:

Analysability:  o       (3.13)
Changeability:  o       (2.95)
Testability:    +       (3.89)

==============================

If next period is Terminal,
the predicted rankings are:

Analysability:  o       (3.12)
Changeability:  o       (2.96)
Testability:    +       (3.91)

==============================
$
```

Listing 6.1: Terminal output of prediction script

6.5.4. VALIDATION

To validate our trained model, we calculate the root mean squared error method (RMSE) and the R-squared method on the 20% test set of the dataset. RMSE calculates the average difference between the actual values and the values predicted by the model, squaring all differences to treat both positive and negative deviations equally. Our model has the RMSE values as listed in table 6.11. The R-squared method measures the percentage of variation in the dependent variable that is explained by the independent variables in the model. The value of the dependent variable ranges from 0 to 1, where 1 means that the model explains the variation perfectly. For our model, the R-squared value is 0.95563, indicating that it explains 95.56% of the variability in the dependent variable.

Characteristic	**RMSE Value**
Analysability	0.1037
Changeability	0.2149
Testability	0.1031

Table 6.11: RMSE Values for trained model

We observe small deviations for the respective characteristics. If the dataset were larger, the deviations would be smaller. We can consider our model to be valid.

7

Discussions

7.1. Reflection on the results

Our research indicates that the maintainability characteristics of open source software do not significantly change with fluctuations in the popularity of a project. We measured this on a dataset of 90 projects over 8 periods of 26 weeks each. However, the popularity of a project does change regularly. During the measurements of our dataset, the OSS community category changed between 51% of the periods. Interestingly, popular attractive projects often end up in a terminal or stagnant OSS community category after several periods. This transition usually does not affect the maintainability characteristics. It seems that after a while, OSS projects enter a stable phase that is not attractive or dynamic enough for new developers, and they eventually rely on their core members.
For projects where no OSS community category transition occurs between periods (49%), we observed that projects in the attractive OSS community category experience changes in the overall maintainability ranking more frequently than average. However, due to the limited number of projects, it is difficult to determine whether it is a trend. A larger dataset could provide better insight into this.

Within the periods measured for our research, there is also the period that covers the COVID-19 pandemic (March 2020). When we zoom in on this period, it is noticeable that there is a drop in attractive and a peak in terminal OSS community categories. This is strange because you would expect the opposite in OSS communities, as OSS developers are usually individuals who contribute from home.
The results of our research differ from the expectations that changes in popularity would have a significant impact on maintainability. This is not the case.

7.2. Reflection on the research

Our research focused on analyzing the relationship between community aspects and maintainability of open source projects, using a dataset of 90 Java projects from GitHub. These projects were randomly selected based on the number of stars to obtain as varied a dataset as possible. Since existing datasets, such as GHTorrent by Gousios [9], were no longer maintained we had to create the dataset ourselves via the GitHub API. A larger dataset could provide more detailed insights, but this was not realistic because data collection via the GitHub API is a time consuming process.

We composed a custom quality model after studying various existing models. We based our model on metrics that were specifically relevant to our research. The calculation of maintainability is mainly based on the SIG maintainability model [10]. However, we did not include the unit tests metric because it is difficult to automate and prone to errors. Unit tests are important and ideally should be included. The SIG maintainability model is an alternative to the maintainability index of Coleman et al. [7] but does not use metrics for comments in the code. We have added the original metric for comments of Coleman et al. to our model and expanded it with an additional metric for the percentage of relevant comments from Misra et al. [14]. For the community aspects in our model, we measured the popularity of a project based on the developer activity in Git commits during a selected period. We used the metrics of Yamashita et al. for this [24]. However, community aspects include more than just developer popularity. Activities such as responding to forum posts, documenting features, or testing functionalities can also impact a community. Although we would have liked to include more community-specific aspects, such as the number of stars and the resolving time of issues, assigning weighted quality rankings proved too complex to accomplish within the available time. We have developed a quality model that is suitable for our research but can still be expanded in the future.
To apply our quality model to our dataset, we developed a tool that can calculate the metrics for each period and each repository. Existing tools, such as SonarQube, were not suitable for automatically measuring a large number of repositories across different periods. It was also complicated to implement the metrics we wanted to use, such as the SIG maintainability model or the metrics of Yamashita et al., in SonarQube [1]. Therefore, we created our own terminal tool, which uses existing tools under the hood. The advantage of a custom developed tool is that it is tailored, which makes it easy to make adjustments in case of changes. A disadvantage is that its development was time consuming. The tool we developed allows us to quickly calculate metrics for selected periods for 90 repositories and store the results in a database or file. The next phase of the research was analysing the results of the measured metrics. Visualizing the maintainability characteristics for specific OSS community categories across various periods was challenging. We addressed this issue by using various types of heatmaps. These heatmaps provided clear insights into the changes in specific characteristics by OSS community category and period. Our other graphs, scatter plots, and tables also provided a clear overview of the measured metrics. These visualizations helped us to analyze and answer our main research question and sub question 3. With the results in our database, we were able to train a linear regression model to use in a prediction tool. With limited knowledge and without conducting in-depth research, we managed it to build a tool using existing Python frameworks by studying some examples on the internet. We were able to answer the optional research sub question 4 here.
Although the dataset and quality model could be improved, I believe we have effectively addressed the research questions.

7.3. LIMITATION

During the research, we encountered several limitations. The GHTorrent [9] tool we wanted to use was unavailable, so we wrote our own GitHub parser, which was time-consuming.
The open source quality model SQO-OSS [19], which we wanted to adapt our model to, was not fully available. For example, the metrics that were used were not described anywhere. Therefore, it was not possible to use this model.

[1] `https://www.sonarsource.com/products/sonarqube/`

The metrics from the SIG maintainability model [10] on unit tests (code coverage and assert statements) were not included in our quality model because they are difficult to automate for so many instances. Therefore, we do not have a stability ranking. However, unit tests are an important aspect of software quality and should ideally be included.
Some metrics use LOC, and as described in section 5.1, there are various techniques for this. In our research, we use the traditional LOC, but it would be better to use the NCSS technique. Unfortunately, this was not possible because not all the external tools we use support the NCSS technique.

7.4. Threats to validity

In section 4.2, we replaced an existing metric on unit testing with a new metric for comments that we composed. The rankings for this new metric were determined by us and are used in assessing the final analysability and changeability rankings. Although we believe the final ranking correctly reflects the data, it is possible that it produces results different from what the original SIG maintainability model intended. Should future work reintroduce the metric on unit testing (see section 7.5) while retaining the comments metric, it will be important to assess the weightings of the rankings.

7.5. Future research

Due to the limited time available for our research, there are still various aspects, especially regarding the OSS community, that could be interesting for future research. The answers and results of our research questions can also provide interesting input for new or more in-depth studies. Using a much larger dataset and a wider range of periods could also provide different insights.

As described in section 7.1, there appears to be a variation in measurements during the COVID-19 pandemic. We observe in figure 6.14 a peak in the terminal OSS state categories and a downfall in the attractive OSS state categories around March 2020. It could be interesting to study this in more detail. During the pandemic, research on this was already done by Wang et al. [23]. According to this study, activities within GitHub communities seem to have increased during the pandemic. However, it is noticeable that the number of new developers also experienced a dip at the end of 2019.
The measurements also indicate that projects often end up in the OSS state categories of terminal or stagnant without the maintainability worsening. This suggests that the modifications to these projects are carried out by a core group of developers who remain active. This could also be interesting to study further.
In the composed quality model (section 4.1), the community aspect could be extended. Currently, only the activity of developers is considered, but for example, a bug reporter or someone who writes documentation is currently not included. Other aspects, such as the metadata from stars, issues or forks, might also provide insights about a community. For the maintainability aspect, as mentioned in section 7.3, the model could be extended to include metrics (code coverage, assert statements) on unit tests, as in the original SIG maintainability model [10].

The dataset we used consists exclusively of Java repositories. The same research could also be conducted with repositories from another programming language. PHP, for example, could be a good choice because many OSS communities are active in it.

As mentioned in section 7.1, it seems that popular attractive projects struggle to maintain high maintainability characteristics. This may be caused by peripheral developers and could be an interesting topic for further research.

BIBLIOGRAPHY

[1] ISO/IEC 9126-1:2001. Standard, International Organization for Standardization, 2001. URL https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/02/27/22749.html. 2, 8, 9, 10

[2] ISO/IEC TR 9126-2:2003. Technical report, International Organization for Standardization, 2003. URL https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/02/27/22750.html. 8

[3] ISO/IEC TR 9126-3:2003. Technical report, International Organization for Standardization, 2003. URL https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/02/28/22891.html.

[4] ISO/IEC TR 9126-4:2004. Standard, 2004. URL https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/03/97/39752.html. 8

[5] Rafa E Al-Qutaish. Quality models in software engineering literature: an analytical and comparative study. *Journal of American Science*, 6(3):166–175, 2010. 7

[6] Youness Boukouchi, Abdelaziz Marzak, Habib Benlahmer, and Hicham Moutachaouik. Comparative study of software quality models. *IJCSI International Journal of Computer Science Issues*, pages 309–314, 2013. 7, 8, 9

[7] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994. 25, 64

[8] Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994. 11

[9] Georgios Gousios. The GHTorent dataset and tool suite. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 233–236. IEEE, 2013. 5, 19, 63, 64

[10] Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*, pages 30–39. IEEE, 2007. 8, 11, 12, 13, 14, 15, 24, 29, 64, 65

[11] TJ McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4): 308, 1976. 12

[12] Jim A McCall, Paul K Richards, and Gene F Walters. Factors in software quality. volume i. concepts and definitions of software quality. Technical report, General Electric Co., Sunnyvale, CA (USA), 1977. 7

[13] Kelvin McClean, Des Greer, and Anna Jurek-Loughrey. Social network analysis of open source software: A review and categorisation. *Information and Software Technology*, 130:106442, 2021. 2, 4, 5

[14] Vishal Misra, Jakku Sai Krupa Reddy, and Sridhar Chimalakonda. Is there a correlation between code comments and issues? an exploratory study. pages 110–117, 2020. 15, 16, 17, 26, 33, 64

[15] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *Proceedings of the international workshop on Principles of software evolution*, pages 76–85, 2002. 4, 5, 6

[16] Paul Oman and Jack Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–338. IEEE Computer Society, 1992. 11

[17] Saya Onoue, Hideaki Hata, Akito Monden, and Kenichi Matsumoto. Investigating and projecting population structures in open source software projects: A case study of projects in github. *IEICE TRANSACTIONS on Information and Systems*, 99(5):1304–1315, 2016. 5

[18] Roger S. Pressman. *Software engineering: a practitioner's approach*. McGraw-Hill, fifth edition, 2001. 12, 13

[19] Ioannis Samoladas, Georgios Gousios, Diomidis Spinellis, and Ioannis Stamelos. The SQO-OSS quality model: measurement based open source software evaluation. In *IFIP international conference on open source systems*, pages 237–248. Springer, 2008. 64

[20] Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality analysis of source code comments. pages 83–92, 2013. 15

[21] Igor Steinmacher, Gustavo Pinto, Igor Scaliante Wiese, and Marco Aurélio Gerosa. Almost there: A study on quasi-contributors in open-source software projects. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 256–266. IEEE, 2018. 2

[22] MR Martinez Torres, Sergio L Toral, M Perales, and Federico Barrero. Analysis of the core team role in open source communities. In *2011 International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 109–114. IEEE, 2011. 2

[23] Liu Wang, Ruiqing Li, Jiaxin Zhu, Guangdong Bai, Weihang Su, and Haoyu Wang. Understanding the impact of covid-19 on github developers: A preliminary study. In *SEKE*, pages 249–254, 2021. 65

[24] Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, and Naoyasu Ubayashi. Magnet or sticky? an oss project-by-project typology. In *Proceedings of the 11th working conference on mining software repositories*, pages 344–347, 2014. i, 2, 3, 5, 6, 18, 24, 35, 36, 39, 41, 42, 64

[25] Kazuhiro Yamashita, Yasutaka Kamei, Shane McIntosh, Ahmed E Hassan, and Naoyasu Ubayashi. Magnet or sticky? measuring project characteristics from the perspective of developer attraction and retention. *Journal of Information Processing*, 24(2):339–348, 2016. 2, 3, 6, 7

[26] Yunwen Ye and Kouichi Kishida. Toward an understanding of the motivation of open source software developers. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 419–429. IEEE, 2003. 2, 4, 5

APPENDIX A - GITHUB PROJECTS

Table 1: GitHub projects in the dataset

No.	Project	Created at	Stars	Forks
1	square/retrofit	2010-09-06	41252	7242
2	apache/dubbo	2012-06-19	38521	25697
3	skylot/jadx	2013-03-18	33651	4211
4	alibaba/nacos	2018-06-15	25656	11404
5	alibaba/fastjson	2011-11-03	25199	6521
6	alibaba/spring-cloud-alibaba	2017-12-01	24966	7655
7	greenrobot/EventBus	2012-07-16	24275	4678
8	seata/seata	2018-12-28	23424	8354
9	google/gson	2015-03-19	21907	4209
10	redisson/redisson	2014-01-11	20746	4988
11	alibaba/Sentinel	2018-04-04	20643	7496
12	termux/termux-app	2015-10-23	19932	2408
13	ReactiveX/RxAndroid	2014-08-19	19767	3009
14	apache/rocketmq	2016-11-30	18865	10632
15	brettwooldridge/HikariCP	2013-10-08	17934	2701
16	Tencent/tinker	2016-09-06	16701	3332
17	LMAX-Exchange/disruptor	2012-09-21	15811	3755
18	hdodenhof/CircleImageView	2014-01-17	14390	3140
19	baomidou/mybatis-plus	2016-08-18	14145	3845
20	alibaba/ARouter	2016-12-14	14117	2538
21	mockito/mockito	2012-10-13	13751	2369
22	GoogleContainerTools/jib	2018-01-22	12591	1356
23	Netflix/zuul	2013-03-13	12479	2278
24	Netflix/eureka	2012-07-26	11730	3674
25	redis/jedis	2010-06-11	10954	3730
26	code4craft/webmagic	2013-04-23	10755	4116
27	square/javapoet	2013-02-01	10101	1312
28	google/auto	2013-05-22	10043	1186
29	jhy/jsoup	2009-12-19	10023	2067
30	perwendel/spark	2011-05-05	9465	1571
31	google/bundletool	2018-05-04	2985	352
32	AxonFramework/AxonFramework	2011-12-02	2971	748
33	halirutan/IntelliJ-Key-Promoter-X	2017-07-28	2937	67
34	JackyAndroid/AndroidTVLauncher	2016-03-27	2932	715
35	hneemann/Digital	2016-06-28	2932	326
36	KeepSafe/ReLinker	2015-10-15	2927	356
37	apache/curator	2014-03-03	2919	1199
38	mbechler/marshalsec	2017-05-22	2915	674

The list continues on the next page

Table 1 – *Continuation from the previous page*

No.	Project	Created at	Stars	Forks
39	microservices-patterns/ftgo-application	2017-10-23	2906	1148
40	Netflix/concurrency-limits	2017-12-11	2896	280
41	igniterealtime/Smack	2014-02-03	2322	887
42	pilgr/Paper	2015-06-11	2321	234
43	assertj/assertj	2013-03-14	2317	610
44	apache/maven-mvnd	2019-09-21	2279	170
45	sofastack/sofa-bolt	2018-04-09	2272	824
46	hierynomus/sshj	2010-02-27	2256	545
47	linkedin/rest.li	2012-11-30	2253	520
48	openzipkin/brave	2013-04-07	2253	707
49	rsocket/rsocket-java	2015-07-08	2247	343
50	google/jimfs	2013-10-21	2246	266
51	FabricMC/fabric	2018-11-04	1660	328
52	selenide/selenide	2012-02-07	1650	537
53	srikanth-lingala/zip4j	2019-05-04	1636	274
54	500px/greedo-layout-for-android	2016-02-05	1634	160
55	exchange-core/exchange-core	2018-08-05	1626	683
56	CaffeineMC/lithium-fabric	2019-11-30	1624	161
57	mc1arke/sonarqube-community-branch-plugin	2019-03-04	1615	399
58	jchambers/pushy	2013-08-02	1604	431
59	aNNiMON/Lightweight-Stream-API	2015-01-01	1602	129
60	OpenHFT/Java-Thread-Affinity	2013-09-20	1592	334
61	reactor/lite-rx-api-hands-on	2016-02-01	988	929
62	jfree/jfreechart	2016-02-01	982	390
63	hub4j/github-api	2010-04-19	982	665
64	jmrozanec/cron-utils	2014-08-08	981	246
65	java-diff-utils/java-diff-utils	2017-03-30	980	157
66	vanilla-music/vanilla	2012-09-22	980	280
67	structurizr/dsl	2020-06-13	977	234
68	mixpanel/mixpanel-android	2010-08-05	976	355
69	wstrange/GoogleAuth	2012-01-10	975	319
70	PortSwigger/param-miner	2018-07-26	972	160
71	Esri/geometry-api-java	2013-01-15	664	247
72	eclipse-californium/californium	2015-09-24	664	354
73	asr-pub/LyricViewDemo	2016-12-26	662	114
74	ppareit/swiftp	2012-02-06	662	284
75	jenkinsci/slack-plugin	2013-11-21	661	412
76	rharter/auto-value-parcel	2015-05-13	661	65
77	yacy/yacy_grid_mcp	2017-01-30	657	26
78	apache/mina-sshd	2010-05-26	657	293
79	swagger-api/swagger-parser	2014-04-21	655	483
80	xuhuisheng/sonar-l10n-zh	2012-10-03	655	216
81	ua-parser/uap-java	2014-11-09	333	161
82	ItzSomebody/radon	2018-01-11	333	78
83	SeeSharpSoft/intellij-csv-validator	2017-09-14	332	42

The list continues on the next page

Table 1 – *Continuation from the previous page*

No.	Project	Created at	Stars	Forks
84	esoco/objectrelations	2015-12-02	10	2
85	fabioz/eclipse.spellchecker	2013-10-22	10	4
86	cyNeo4j/cyNeo4j	2014-02-12	10	11
87	farnulfo/pst-exp	2015-08-26	2	4
88	indigo-dc/im-java-api	2015-10-29	2	0
89	ronnypolley/ecliseignorehelper	2015-12-08	2	0
90	imagej/imagej-troubleshooting	2015-11-18	2	3

APPENDIX B - DATABASE STRUCTURE

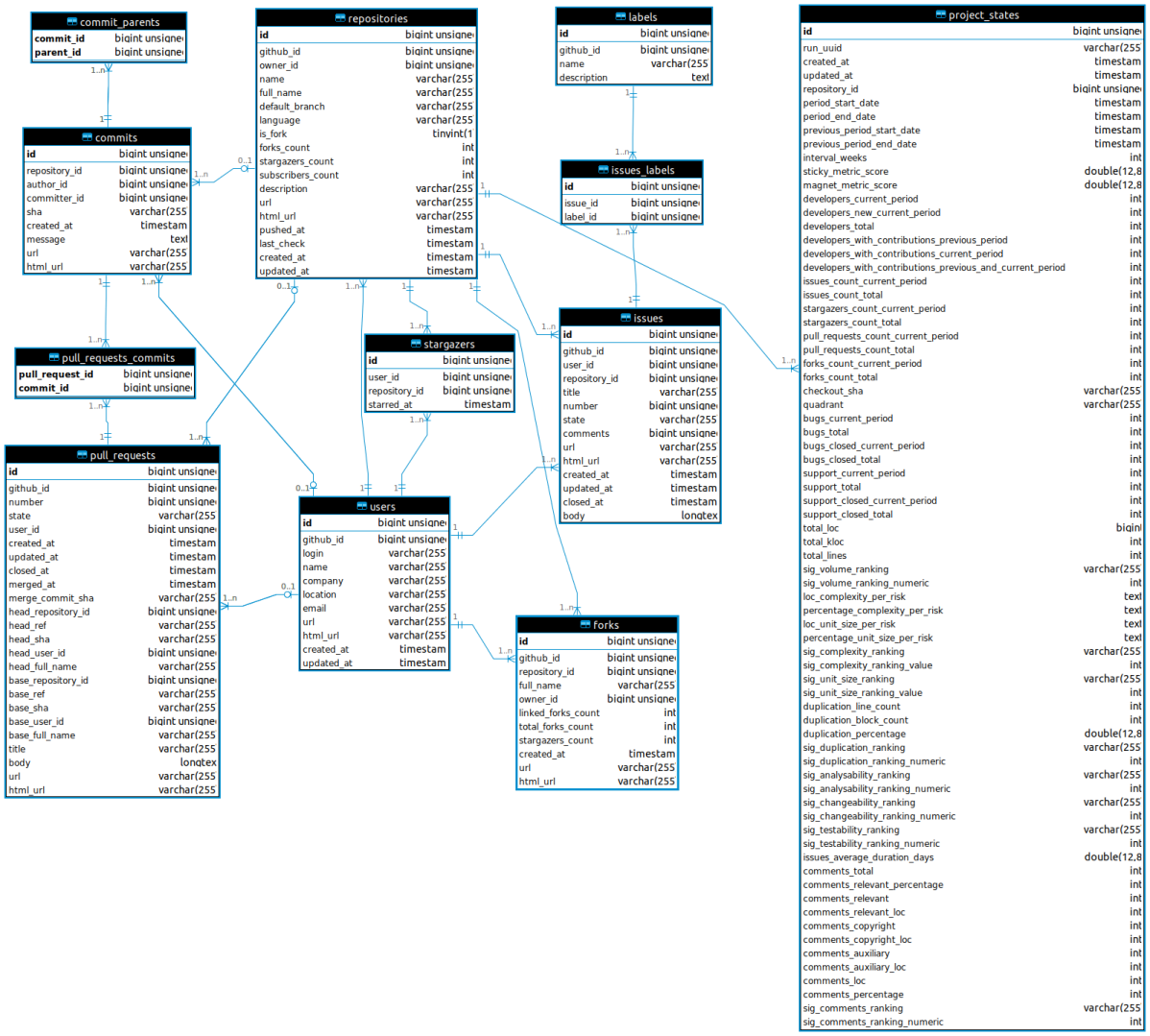


Figure 1: MySQL database structure of dataset and measurement results