2018 IEEE International Conference on Software Quality, Reliability and Security

# How Do Defects Hurt Qualities? An Empirical Study on Characterizing A Software Maintainability Ontology in Open Source Software

Celia Chen*¶, Lin Shi†§, Michael Shoga¶, Qing Wang†‡§,Barry Boehm*
*Center for Systems and Software Engineering, University of Southern California, Los Angeles, USA
Email:{qianqiac, boehm}@usc.edu
†Laboratory for Internet Software Technologies, Institute of Software Chinese Academy of Sciences, Beijing, China
‡ State Key Laboratory of Computer Sciences, Institute of Software Chinese Academy of Sciences, Beijing, China
§University of Chinese Academy of Sciences, Beijing, China
Email: {shilin, wq}@itechs.iscas.ac.cn
¶Occidental College, Los Angeles, CA
Email: {shoga}@oxy.edu

***Abstract*—Beyond the functional requirements of a system, software maintainability is essential for project success. While there exists a large knowledge base of software maintainability, this knowledge is rarely used in open source software due to the large number of developers and inefficiency in identifying quality issues. To effectively utilize the current knowledge base in practice requires a deeper understanding of how problems associated with the different qualities arise and change over time. In this paper, we sample over 6000 real bugs found from several Mozilla products to examine how maintainability is expressed with subgroups of repairability and modifiability. Furthermore, we manually study how these qualities evolve as the products mature, what the root causes of the bugs are for each quality and the impact and dependency of each quality. Our results inform which areas should be focused on to ensure maintainability at different stages of the development and maintenance process.**

***Keywords*—software maintainability, software maintainability ontology, software quality, software quality measurement**

## I. Introduction

While the primary functionality of a software system is often the main focus, being highly maintainable is essential to ensure that the product satisfies performance, cost, and dependability objectives. With the increasing rate of change in technology, competition, organizations, and the marketplace[1], a majority of most software systems' life cycle cost has been spent in software maintenance. Koskinens 2009 survey[2] found that 75-90% of business and command&control software and 50-80% of cyber-physical system software costs are incurred during maintenance. However, defects and bugs in software hurt maintainability, resulting in problems for developers and consumers as well as affecting the primary functionality. Despite the importance of maintainability in project development, lack of information regarding where these issues occur and how these problems evolve over time impede their resolution.

* Lin Shi is the corresponding author.

To aid potential adopters of existing software components to be reused and evolved in practice, a number of software maintainability metrics have been developed [3]. Most of these involve automated analysis of the code, such as Maintainability Index [4], technical debt [5][6], code smells [7][8][9], and other Object-Oriented metrics [10][11][12]. There are also reuse cost models that estimate maintainability of potentially reusable components based on human-assessed maintainability aspects, such as, code understandability and structure [13][14].

While these automated methods are easy to use, Riaz *et.al* [15] pointed out that the effective use of accuracy measures for these metrics has not been observed and there is a need to further validate maintainability prediction models. Moreover, they only identify problematic areas but do not provide an overall quality status for the current version. Problem identification is valuable; however, a stronger and more thorough understanding of how defects detract from software qualities (SQs) is necessary for the development of software with high SQs. On the other hand, human-assessed methods can more accurately reflect the actual maintenance effort spent in maintenance tasks, yet it is too expensive and subjective to collect throughout software development[13][16]. Especially in open source software, since developers aren't required to sign any contracts before joining, personnel turnover is extremely high. Consequently, the development effort invested in maintaining an open source project is usually unknown, even after the software has been released[17]. Additionally, most of the projects do not have any official hiring process to select developers with the appropriate skills and experience prior to contributing, thus making maintainability difficult to measure.

In this paper, we examine how a maintainability ontology, focusing on the supporting qualities of repairability and modifiability can be expressed in bug reports found in 11 products from the Mozilla community. Specifically, this study aims at investigating how maintainability changes as software evolves,

978-1-5386-7757-5/18/$31.00 ©2018 IEEE
DOI 10.1109/QRS.2018.00036

the maintainability differences under various circumstances, and how the subgroup SQs of the maintainability ontology relate to each other. We mine 61790 bugs and manually analyze 6372 of them. To answer these research questions, we first identify and map each bug to its expressing subgroup SQ. Then we group them by the life cycle, software domain, severity and fix time to analyze the characteristics of maintainability issues. Furthermore, we extract bugs that depend on other bugs to study the dependency relationship between the SQs the bugs express. Our quantitative and qualitative results inform the areas in which should be focused on to ensure software maintainability at different stages of the software life cycle and suggest the possibility of automating the process to provide a more thorough understanding of maintainability status for software systems.

The rest of this paper is organized as follows. Section II describes the design of our study. Section III reports the analysis of the results. Section IV states the threats to validity. Section V discusses the results and provides several valuable findings for the research community. Section VI summarizes related work and presents the differences of those compared to our study and Section VII concludes the study.

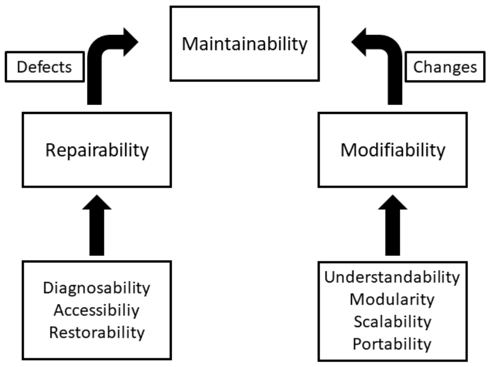
## II. Study Design

The goal of the study is to analyze how a software maintainability ontology is expressed in open source software systems, with the purpose of understanding how these qualities evolve as software evolves. Since bugs are used widely in open source software as an indication of quality [18][17], in this paper, we examine how maintainability issues are expressed in reported bugs. The definition of our terminology came from existing literatures. A "bug" is a synonym of a "fault", which means "an event that occurs when the delivered service deviates from correct service" [19]. In the rest of this paper, we use the general name "bugs" to refer to resolved and fixed bugs.

The study aims at addressing the following three research questions:

- RQ1. *As software evolves, how does software maintainability change?* This research question aims at investigating to what extent the common wisdom suggesting "Declining Quality"[1][20][21] applies in the context of maintainability. Specifically, we study how the percentage of maintainability issues changes, to understand whether software maintainability rises, or whether it decreases as software grows mature and stable. To this aim, we further investigate the dominant subgroup SQs contributing to software maintainability, to understand which subgroup SQs tend to have more issues in the earlier phase, middle phase and later phase of the life cycle. Moreover, we point out the root causes of the bugs that express these SQ issues and whether these root causes remain the same, or rather change as software evolves.
- RQ2. *How are software maintainability issues expressed in different classifications of bugs?* This research question aims at exploring under which circumstances maintainability issues are more prone to be introduced. More specifically, we focus on studying how maintainability issues are affected by bugs with different severity levels and fix times. We also look into the influence of reopened bugs on maintainability and its subgroup SQs. In addition, the maintainability differences between client software and server software are examined.
- RQ3. *How do the subgroups of the maintainability SQs relate to each other?* We aim to examine the relationship among the subgroup SQs through bug dependency. Bugs that affect one SQ can introduce additional bugs in the same or a different SQ. We focus on finding dependency relationships in the types of SQs that depend on each other as well as identifying root causes along the chain.

Fig. 1: Maintainability Hierarchy



### A. Software Maintainability Ontology

Boehm, *et. al*[22][23] provided an IDEF5 class hierarchy of upper-level SQs, where the top level reflected classes of stakeholder value propositions (Mission Effectiveness, Resource Utilization, Dependability, Flexibility), and the next level identified means-ends enablers of the higher-level SQs. In pursuing the various contributions and types of maintainability as a critical but complex SQ, they have elaborated on various maintainability relationships.

Their hierarchy defines maintainability as an external support to the SQ changeability [24]. Maintainability depends on two alternative SQs, repairability and modifiability which handle defects and changes respectively. These SQs are further enabled by several subgroups. Our study focuses on maintainability in the context of these mean-ends SQs as shown in Fig.1.

Repairability[23] involves handling of defects in software. It is enabled by the following SQs:

- *Diagnosability*[25]: involves error messages and the process of tracing where they originate. Bugs that affect this SQ involve problems with log messages, failure of tests, and insufficient information provided for accurate assessments.
- *Accessibility*[26]: involves whether or not a user is able to access intended areas of software. Bugs that affect this SQ prevent the user from accessing data or functions due to things such as redirects to unintended locations, error messages, and crashes.

TABLE I: Root Causes

| **Internal Implementation** | **Interface** | **External** |
|---|---|---|
| Syntax/Semantic Errors | Interface Semantic Errors | External Semantic Errors |
| Data Design/Usage | Functionality Design/Usage | Development/ Deployment Environment |
| Resource Allocation/Usage | GUI Design/Usage | External Tools/ Infrastructure |
| Exception Handling | Unexpected Interactions | Test Cases |
| Functionality | | Previous Releases |
| Performance | | |
| Documentation/ Comments | | |

- *Restorability*[27][28]: involves whether the software is able to restore to a previous state. Bugs that affect this SQ prevent activities such as clearing of caches, refreshing settings, and proper removal of data.

Modifiability [29] involves handling of software changes. It is enabled by the following SQs:

- *Understandability* [14]: involves how easily understood software is. Bugs that affect this SQ involve lack of explanations or comments, confusing or inaccurate descriptions, or presence of deprecated software.
- *Modularity* [30]: involves separation of code into modules. Bugs that affect this SQ involve unwanted interactions between different modules.
- *Scalability* [31]: involves decreases in performance due to changes of software. Bugs that affect this SQ involve latency in functionality, hangs and crashes, and insufficient resources for functionality.
- *Portability* [32]: involves independence of hardware, OS, middleware, and databases. Bugs that affect this SQ prevent proper interfacing between software components and external platforms.

## B. Root Causes

For each subgroup SQ, we also study the root causes of the bugs that express that particular SQ issue. We designed and defined the list of root causes from existing literatures [33][34][35][36] and our experiences of bugs in other projects. Table I shows the descriptions of the root cause categories and their sub-categories we defined for this study.

The root causes for our study are divided into three classes: internal implementation, interface, and external. Broadly speaking, these classes cover issues found in situations such as how the software was implemented, how users will interact with the software, and how the software interacts with external software.

The root causes within internal implementation are as follows:

- *Syntax/Semantic Errors*: Syntax errors indicate problems that violate the grammatical rules of the programming languages, while semantic errors reflect improper use of the programming languages. We include typos into this category to simplify the categories. This type of root cause is found in all three classes. Examples within internal implementation include missing punctuation or using non-initialized variables.
- *Data Design/Usage*: involves problems with how data is stored, organized, and accessed.
- *Resource Allocation/Usage*: involves problems with how resources are handled such as how much memory is allocated and interactions between software components.
- *Exception Handling*: involves improper handling of thrown exceptions.
- *Functionality*: involves cases in which the primary functionality of the software is not possible.
- *Performance*: involves problems in which the software has latency problems, hangs, or crashes.
- *Documentation/Comments*: involves incomplete or unclear documentation or comments for the software.

Root causes within the interface class are as follows:

- *Interface Semantic Errors*: an example of semantic errors found in this class is incorrectly extending an interface from another parent class.
- *Functionality Design/Usage*: involves problems with the main functionality of the GUI components such as broken buttons.
- *GUI Design/Usage*: involves problems with GUI such as visual problems or improperly displayed messages.
- *Unexpected Interactions*: involves problems with incorrect handling of actions between GUI components.

Root causes within the external class are as follows:

- *External Semantic Errors*: external semantic errors can be expressed as an incorrect reference to an external package or library.
- *Development/Deployment Environment*: involves problems restricted to specific platforms/environments, for example in a particular OS.
- *External Tools/Infrastructure*: involves problems with software and tools that are not part of the original project, such as external APIs and libraries.
- *Test Cases*: involves problems with the particular test cases.
- *Previous Releases*: involves problems that persist from older releases or software that requires updating.

## C. Study Subjects

The context of our study consists of the analysis of the changes of maintainability along with its subgroups in 11 products found in the Mozilla software ecosystem. Table II reports the characteristics of each product: (i) the classification of the product, (ii) earliest bug reported, and (iii) the number of sampled bugs. We limit our study to resolved and fixed bugs since unfixed bugs may be invalid and the root causes cannot be identified through bug reports and follow-up discussions. To sum up, we mine 61790 bugs; we sample and analyze in total 6372 bugs. All the bug reports of the products are hosted on Bugzilla[1].

[1]https://bugzilla.mozilla.org/

TABLE II: Characteristics of study subjects

| Classification | Product | Earliest bug | Sampled bugs # |
|---|---|---|---|
| *Client Software* | Cloud Services | 2007 | 617 |
| | Data Platform and Tools | 2015 | 158 |
| | Firefox for Android | 2009 | 807 |
| | Firefox OS | 2011 | 1000 |
| | Thunderbird | 2000 | 707 |
| | SeaMonkey | 1998 | 875 |
| *Server Software* | Bugzilla | 1998 | 841 |
| | Socorro | 2007 | 448 |
| | Webtools | 1999 | 308 |
| | Testopia | 2006 | 130 |
| | Marketplace | 2011 | 481 |

Mozilla was chosen because of the popularity and diversity of its products. The choice of the products to analyze is not random, but rather driven by the motivation to consider products having (i) different domains, *e.g.*, we consider a mix of client applications and server software, (ii) different architectures, *e.g.*, we have Android mobile apps, development tools, and libraries, and (iii) different durations, *e.g.*, some products have been around for more than ten years whereas some have only been around for less than three years.

### *D. Data Extraction and Analysis*

First we download the resolved and fixed bugs of each product from Bugzilla. Then we randomly select 10% of the bugs from each product as the study subject without replacement. For products that have less than 1000 bugs, we increase the sample size to 30% to have samples with similar scales to those of products with more than 1000 bugs.

Next, we tag each bug with one or more SQs and identify its root cause. For any bugs that express multiple quality issues, we take the first identified SQ tag. Bugs are tagged according to the definitions of each SQ as described in Section II.A. To ensure the quality of the tagging results, we built an inspection team of four experts in software quality ontology and five members from the Mozilla community. All of them either have done intensive research work in software maintainability or have been actively contributing to one or more products in Mozilla community. All the tagging results were reviewed and confirmed by at least four other team members. We hosted discussion sessions to vote for those bugs that received different tagging results.

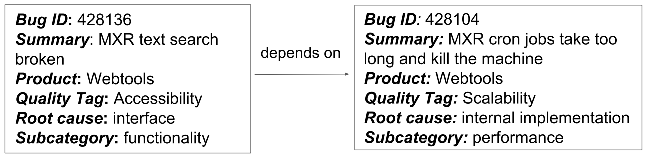
In order to show that we have sampled enough bugs and to indicate the random sampling error in our study, we calculate and present the statistical margin of error with a 95% confidence level for the percentage of maintainability issues. The margin of errors is reported as part of a range of $x \pm i$, which means that if we obtain a value $x$ in a random sample, the actual value of $x$ in the entire population will fall in the range of $[x - i, x + i]$ with a probability of 95%.

To answer RQ1, we use the earliest bug found in each product as the starting point of the product. Then we treat each year as a period and group sampled bugs into different periods.

TABLE III: Characteristics of sampled bugs by severity

| Severity Level | # of bugs |
|---|---|
| *Blocker* | 397 |
| *Critical* | 651 |
| *Major* | 663 |
| *Others* | 4661 |

Fig. 2: A quality dependency relationship example



**Bug ID**: 428136
**Summary**: MXR text search broken
**Product**: Webtools
**Quality Tag**: Accessibility
**Root cause**: interface
**Subcategory**: functionality

**Bug ID**: 428104
**Summary:** MXR cron jobs take too long and kill the machine
**Product:** Webtools
**Quality Tag:** Scalability
**Root cause:** internal implementation
**Subcategory:** performance

To answer RQ2, we first group the sampled bugs by severity. Table III shows the groupings of severity with the corresponding number of bugs. We use the severity label of each bug found on Bugzilla and group them into one of the four severity groups: *blocker*, *critical*, *major*, and *others* in descending order of severity. The *others* group includes bugs that are labeled normal, minor, enhancement, trivia, and others.

We then group the sample bugs by the fix time. The fix time of a bug is calculated using the difference in days between the creation of the the bug and the closing of the bug. For all the sampled bugs, we sort them by the fix time and take 10% of both tails to represent quick-fixed and slow-fixed bugs. In total, we extract 637 bugs for each type. Quick-fixed bugs take 0.29 days while slow-fixed bugs take 3029.19 days on average.

Moreover, we examine reopened bugs in the Mozilla products chosen for our primary analysis. In comparison to the number of resolved bugs, the number of reopened bugs is much smaller. All bugs are examined, and their SQs and root causes are identified. A total of 494 reopened bugs are found and 168 are identified as related to maintainability.

To answer RQ3, we extract all the bugs that depend on other bugs. There are in total 1659 bugs that have some number of dependency relationships. Among those, 439 express maintainability issues, which are considered as the subject for this RQ. We use the term "dependent bug" to refer to the extracted bug and the term "source bugs" to refer to the bugs the dependent bug depends on. The number of source bugs varies from 1 to 91. Most of the dependent bugs depend on one source bug and very few depend on more than six source bugs.

We then generate a quality dependency relationship for each bug with its source bugs. Fig. 2 shows an example of a bug found in Webtools with its source bug. The SQ tag the bug received is accessibility. There is only one source bug in this case, which is found also in Webtools with a SQ tag of scalability. Therefore, the quality dependency relationship generated from this example is: *Accessibility* - $\{$*Scalability*$\}$. We also record the root causes and the subcategories for each relationship. In total, we generate 139 distinct quality relationships.

## III. Analysis of the Results

This section reports the analysis of the results achieved aiming at answering our three research questions.

## A. *RQ1*

Fig. 3 shows the changes in maintainability issues found in each product over time. Each data point represents the percentage of maintainability issues for each period. The linear regression line shows the trend of how maintainability changes as software evolves. The shadow area by default includes the 95% confidence region. As we can observe, all of the client products display an obvious declining trend of maintainability issues as software evolves, which may indicate that as software becomes mature and stable, the maintainability of client software increases. Furthermore, we can see that products in the server software grouping are displaying a slight increase in maintainability issues as software evolves. This may indicate that server software tends to suffer in maintainability as software matures.

We further investigate the dominant subgroup SQs contributing to maintainability during different phases of the software life cycle per product. The actual contribution of each SQ varies among products due to the different characteristics of these products. Fig. 4 shows the trend of accessibility in client products (in Fig. 4(a)) and understandability in server products during phases of software maintenance (in Fig. 4(b)).

We can observe that across all the client products, the percentage of accessibility issues decreases as software evolves. The dominant common root cause of the accessibility issues in the earlier phase shown in Fig. 5 mainly occurs internally, with the specific cause of data design/usage and functionality. This root cause remains and persists until the later phase but is no longer dominant. As software evolves, the root causes of the accessibility issues start shifting from design to more visual and semantic errors. The dominant common root cause of the accessibility issues becomes syntax/semantic errors and interface GUI problems.

Additionally, all products in server software display an increasing trend of understandability issues as software evolves. Internal documentation/comments is consistently the dominant root cause across all three phases.

*Summary for RQ1.* As software evolves, products in client software exhibit an overall increasing trend of maintainability while products in server software show an opposite trend. The dominant subgroup SQs differ among each product for each development phase. However, client products all display a declining trend of accessibility issues while an increasing trend of understandability issues for server products. Moreover, the primary root causes of these dominant SQs exhibit different trends. For accessibility issues, internal implementation problems dominates during the early phase and persists throughout the whole development, however, syntax/semantic and GUI errors start becoming dominant during the later phase. Internal documentation/comments problems persist throughout the software evolution for understandability issues in server products.

## B. *RQ2*

• *Severity.* Fig. 6 shows the percentage of maintainability issues by bug severity. As we can observe, there are $40.81 \pm 4.8\%$ maintainability issues in *blocker* bugs, indicating that we are 95% confident that in the entire Mozilla community, the portion of *blocker* bugs that display maintainability issues is between $40.81 - 4.8\%$ and $40.81 + 4.8\%$. Similar analysis applies to the other severity types. The result indicates that it is statistically significant to suggest that among all the severity levels, *blocker* bugs contain the most maintainability issues. In other words, the most significant bugs in software projects reflect maintainability issues of the software the most. The results also reveal that the majority of maintainability issues are introduced by the high severity bugs, which suggest and validate the importance of maintainability.

Fig. 7 shows how the subgroups of maintainability are expressed in different bug severity types. The percentage of each SQ is normalized for the purpose of comparison. Portability contributes the most in *blocker*, accessibility issues occurs the most in *critical* and *major* while understandability contributes the most to *others*.

Table IV presents the top root cause for each SQ in the subgroups of maintainability by severity types. The results show that most of the SQ issues that are caused by high severity bugs are from internal implementation and external errors. However, *others* bugs triggered SQ issues that are mainly caused by interface and syntax/semantic errors.

• *Fix Time.* Fig. 8 shows how the subgroup SQs are expressed in quick-fixed and slow-fixed bugs. The percentage of maintainability issues shows no statistically significant difference with $31.40\pm3.6\%$ for quick-fixed and $31.08\pm3.59\%$ for slow-fixed bugs.

However, the subgroup SQs contribute differently. As shown in Fig. 9, quick-fixed bugs have mostly understandability issues while portability issues dominate in slow-fixed bugs. Table V further elaborates the most common root cause with its subcategory of each SQ. The kinds of bugs that trigger understandability issues in quick-fixed bugs include internal semantic problems. Portability issues in slow-fixed bugs are mainly caused by problems specifically to a development environment or a target deployment environment externally.

• *Reopened Bugs.* Table VI shows the number and percentage of reopened bug reports followed by their root causes for each subgroup SQ. Accessibility issues make up the majority of reopened bugs with one more bug than the number of understandability bugs. Internal implementation problems account for the top root cause for both accessibility and understandability. Specifically, the root causes are data design/usage and functionality for accessibility and documentation/comments for understandability. Reopening of these bugs indicates that the root causes were not originally fixed or the fixes were insufficient to prevent bugs from recurring.

• *Domain classification.* Fig. 10 illustrates the percentage of the maintainability issues between client and server software. The data shows that there are $29.71 \pm 1.39\%$ maintainability issues in client software and $24.15\pm1.79\%$ in server software, thus suggesting a statistically significant difference between client software and server software.

Further analysis of the dominant subgroup SQ is shown in

Fig. 3: Changes in maintainability issues over time per product, grouped by domain classification

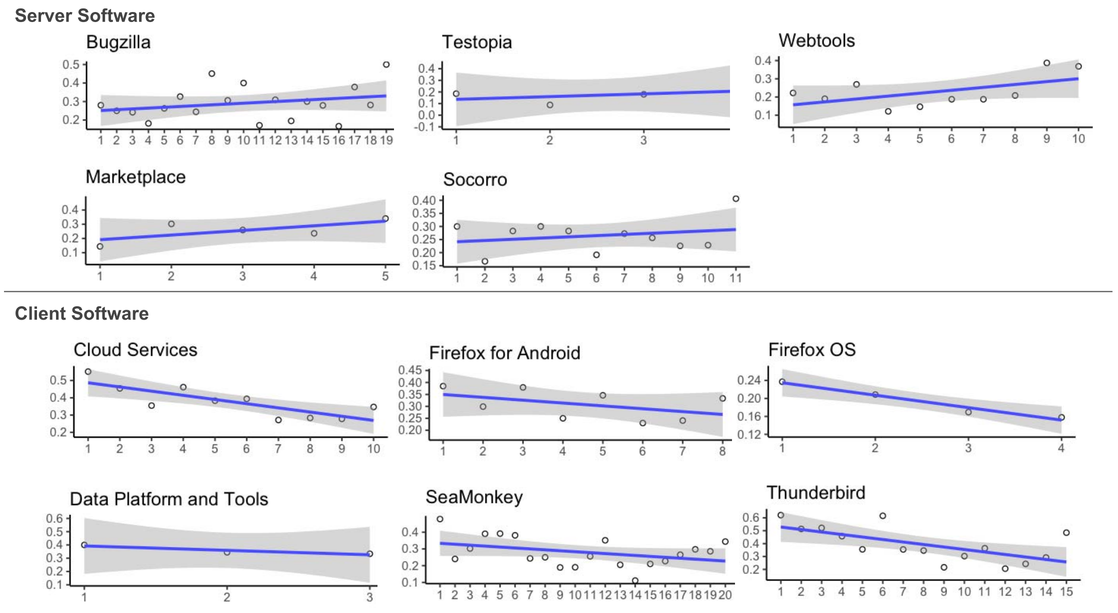


TABLE IV: Root causes of SQs by severity

| | Accessibility | Diagnosability | Modularity | Portability | Restorability | Scalability | Understandability |
|---|---|---|---|---|---|---|---|
| *Blocker* | Performance | Exception handling | Resource allocation/usage | Development/deployment environment | none | Development/deployment environment | Resource allocation/usage |
| *Critical* | Test cases | Exception handling | Internal functionality | External tools/infrastructure | Exception handling | Resource allocation/usage | Documentation /comments |
| *Major* | GUI | Unexpected interaction | Performance | External tools/infrastructure | Interface functionality | Performance | Resource allocation/usage |
| *Others* | Interface functionality | GUI | GUI | GUI | Resource allocation/usage | Performance | Internal semantic errors |

TABLE V: Root causes of subgroup SQs by bug fix-time

| Type | SQ Issue | Root Cause | Root Cause Subcategory |
|---|---|---|---|
| quick-fixed | Understandability | internal implementation | Semantic Errors |
| | Accessibility | interface | GUI |
| | Portability | interface | GUI |
| | Diagnosability | interface | Unexpected interactions |
| | Modularity | interface | GUI |
| | Scalability | interface | GUI |
| | Restorability | N/A | N/A |
| slow-fixed | Portability | external | Development deployment environment |
| | Understandability | internal implementation | Resource allocation /usage |
| | Modularity | internal implementation | Resource allocation /usage |
| | Accessibility | internal implementation | Functionality |
| | Scalability | internal implementation | Performance |
| | Diagnosability | internal implementation | Exception handling |
| | Restorability | internal implementation | Performance |

TABLE VI: Characteristics of SQs of Reopened Bugs

| SQ Issue | % | Root Cause | Root Cause Subcategory |
|---|---|---|---|
| Accessibility | 22.02 | Internal Implementation | Data Design/Usage and Functionality |
| Diagnosability | 13.10 | Interface | Unexpected Interactions |
| Modularity | 7.74 | Internal Implementation | Resource Allocation/Usage |
| Portability | 17.86 | External | External Tools/ Infrastructure |
| Restorability | 1.19 | Internal Implementation | Data Design/Usage and Resource Allocation/Usage |
| Scalability | 16.67 | Internal Implementation | Performance |
| Understandability | 21.43 | Internal Implementation | Documentation/ Comments |

Fig. 12. Understandability is the dominant subgroup SQ for both client and server software.

*Summary for RQ2.* Higher severity bugs express more maintainability issues compared to lower severity, which validates the importance of software maintainability measurement. Quick-fixed and slow-fixed bugs do not express a significant difference in the percentage of maintainability issues. How-

Fig. 4: Relative changes of accessibility and understandability in client software and server software per product respectively

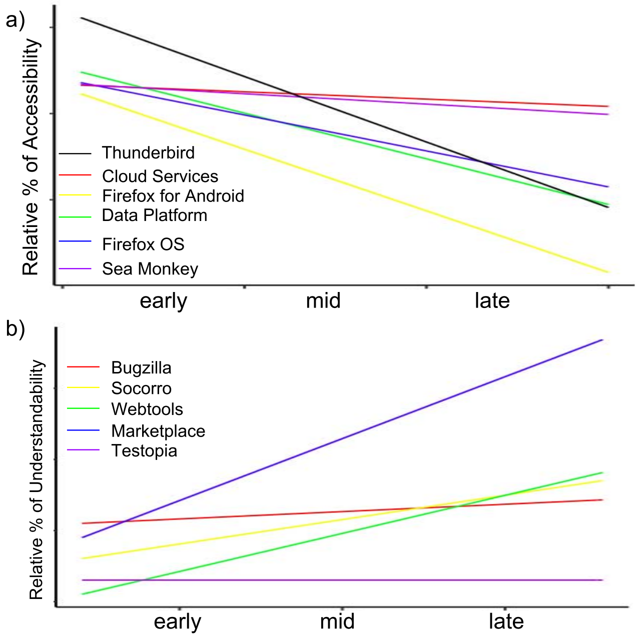


Fig. 5: Distribution of root causes of accessibility issues in client software over time

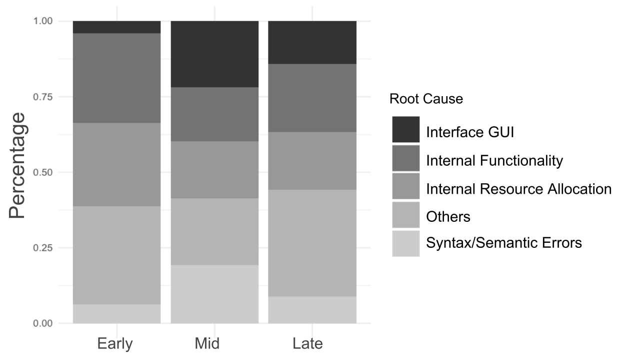


Fig. 6: Percentage of maintainability issues by bug severity

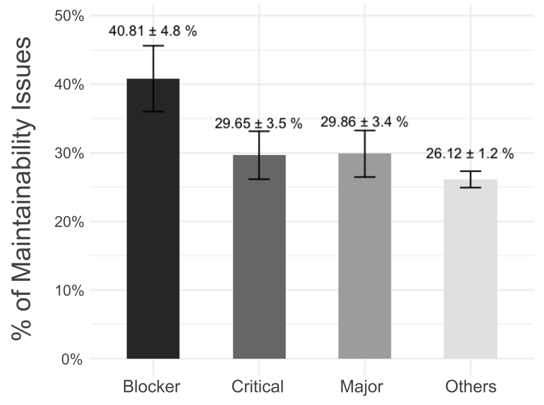


Fig. 7: Percentage of maintainability subgroups SQ by bug severity

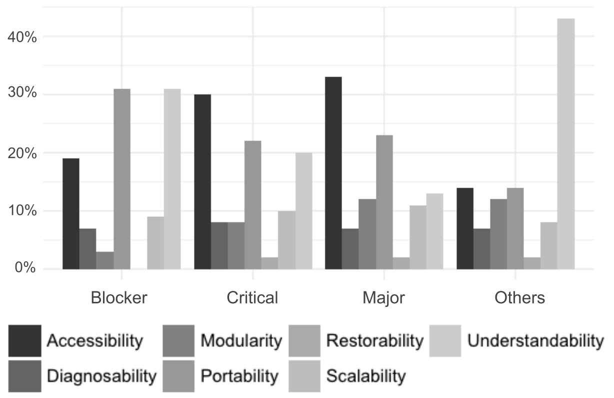
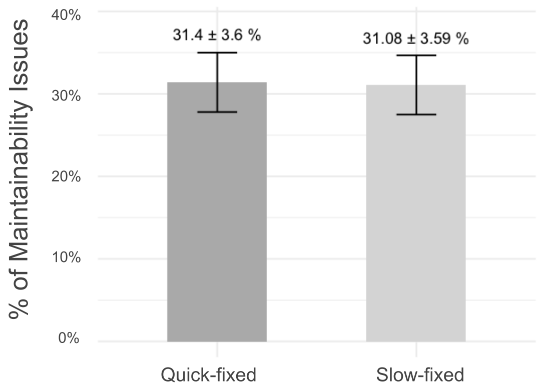


Fig. 8: Percentage of maintainability issues by bug fix time



ever, understandability issues dominate in quick-fixed bugs while portability issues dominate in slow-fixed bugs. Accessibility and understandability issues hold the largest percentage of reopened bugs, with both having internal implementation root causes. Moreover, client software express more maintainability issues compared to server software. Together with the results from RQ1, these results may indicate that when measuring maintainability, it is reasonable to consider different metrics for software in different domain classifications.

## C. *RQ3*

Figs. 11 and 13 illustrate the characteristics of the source bugs in each subgroup SQ. As we can observe, accessibility issues are mostly triggered by SQ issues within the same product while modularity and portability issues are triggered by SQ issues in different products. Modularity issues have the highest number of source bugs. The results may indicate that when encountering an accessibility issue, it is reasonable to suggest searching within the same product. Modularity issues are more likely to be triggered by a composition of SQ issues that are located in different products.

Fig. 14 illustrates the top quality dependency relationships found in the study. The size of each node represents the

Fig. 9: Percentage of maintainability subgroups SQ by bug fix time

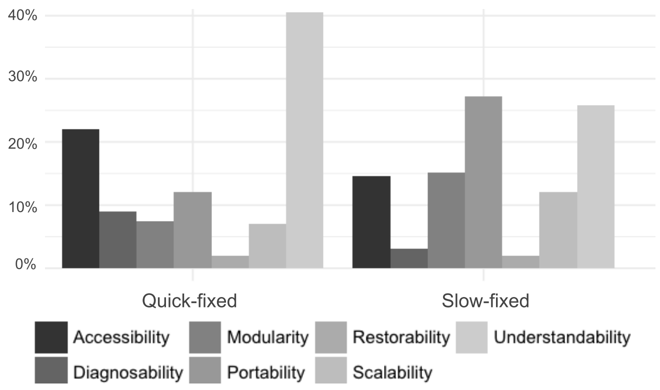


Fig. 10: Percentage of maintainability issues by domain classification

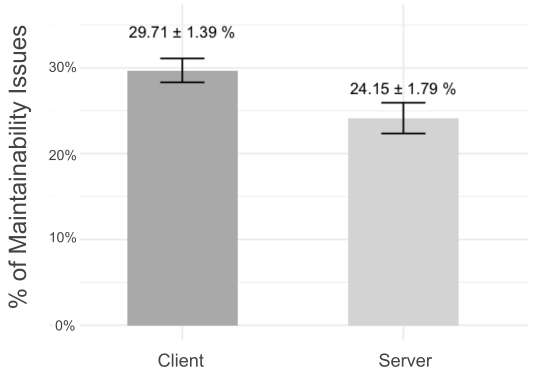


Fig. 11: Average number of source bugs per bug per subgroup SQ

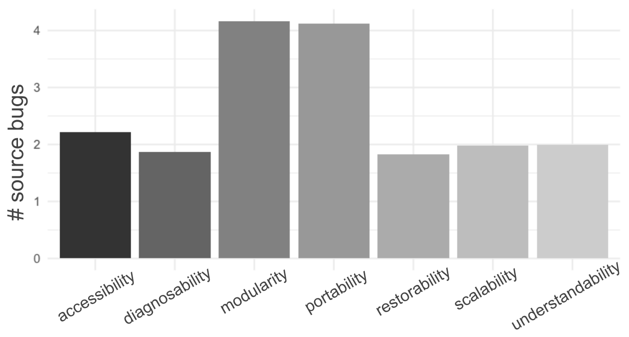


Fig. 12: Percentage of maintainability sub group SQ by domain classification

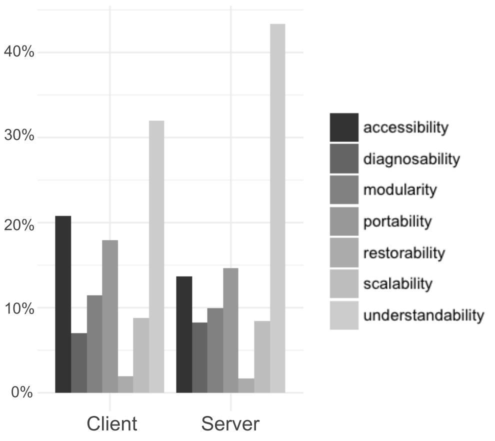
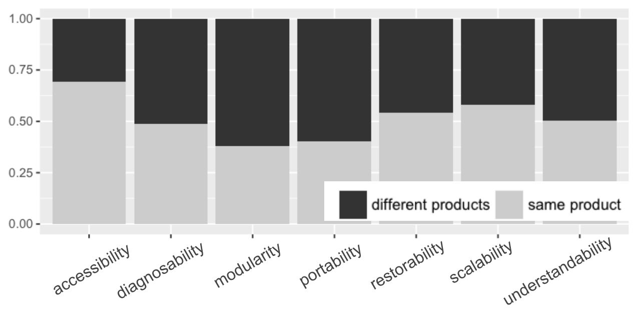


Fig. 13: Percentage of the source bugs found in the same product and different products per subgroup SQ



number of relationships found in each SQ. The bigger the node is, the higher the number of relationships identified for the particular SQ. Understandability contains the highest number of relationships. Since non-maintainability issues are not part of the study goal, the size of the number of relationships found is unknown, thus, no node is shown for it in the graph. The arrow represents the ”from-to” dependency relationship between subgroup SQs. Accessibility and portability contain a self-loop, which indicates that we have found accessibility issues that depend on other accessibility issues. Same applies to portability. Accessibility and non-maintainability issues are the most depended on SQs among these top relationships. Table VII lists the characteristics of the top ten quality dependency relationships. The most occurring relationship is *understandability* $< - \{accessibility\}$. This can occur when data or resources are modified, causing the accessibility issue; this can lead to requests for documentation changes that do not accurately reflect the structure. Non-maintainability issues were found to lead to portability and scalability issues in terms of performance.

*Summary for RQ3.* A list of quality dependency relationships are identified. Accessibility issues are mostly dependent on SQ issues within the same product. Modularity issues have the highest number of source bugs and mostly depend on bugs within different products. The most frequent relationship originates with accessibility issues, which lead to understandability issues. These results provide additional information for bug localization and searching additional issues that may need to be addressed.

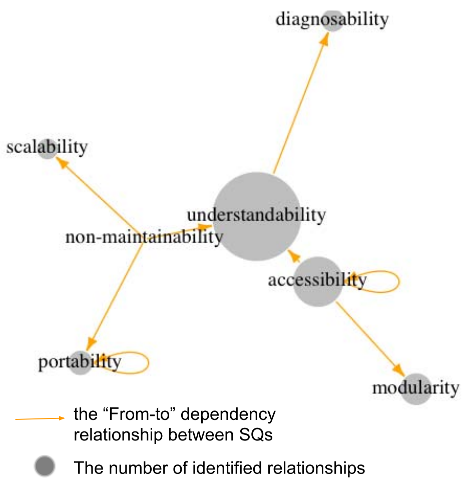
## IV. Threats to validity

While we believe that the bugs from the examined open source software well represent bugs in many software projects,

TABLE VII: Top ten quality dependency relationships alphabetically ranked

| **Quality Dependency Relationship** | **Dependent Bug Root Cause** | **Root cause Subcategory** | **Source Bug(s) Root Cause** | **Root cause Subcategory** | **% of the occurence within each SQ** | **% of the occurrence with the number of bugs with dependency** |
|---|---|---|---|---|---|---|
| *accessibility* <- {*accessibility*} | interface | GUI | interface | functionality | 30.56% | 5.01% |
| | interface | functionality | internal implementation | functionality | 9.72% | 2.96% |
| *diagnosability* <- {*understandability*} | internal implementation | functionality | internal implementation | semantic errors | 35.48% | 2.51% |
| *modularity* <- {*accessibility*} | internal implementation | resource allocation/usage | internal implementation | functionality | 25.93% | 3.19% |
| *portability* <- {*portability*} | external | development /deployment environment | external | previous releases | 17.72% | 3.19% |
| *portability* <- {*Non-maintainability*} | internal implementation | performance | N/A | | 29.11% | 5.24% |
| *scalability* <- {*Non-maintainability*} | internal implementation | performance | N/A | | 40.48% | 3.87% |
| *understandability* <- {*accessibility*} | internal implementation | documentation /comments | internal implementation | data design/usage | 22.82% | 7.74% |
| | internal implementation | documentation /comments | internal implementation | resource allocation/usage | 20.13% | 6.83% |
| *understandability* <- {*Non-maintainability*} | internal implementation | resource allocation/usage | N/A | | 19.46% | 6.61% |

Fig. 14: Quality dependency graph with the number of identified relationships for each quality represented with the size of the node



we do not intend to draw any general conclusions about bugs in all software. Similar to any characteristic study, our findings should be considered together with our evaluation methods. All the products included in this study are from the same open source community. Therefore, our results may not generalize to commercial software or software in other open source communities.

Since we randomly sample bug reports from Bugzilla databases, the distribution of qualities may be different in the entire Bugzilla databases from that of our sample. However, the calculated margin error of 95% confidence level and the reported margins of error give us confidence that the disagreement would be low.

Since we examine bug reports manually, subjectivity is inevitable. However, we try our best to minimize such subjectivity through double verification: each bug report is examined and tagged by at least four team members independently. If the tagging results are different, we host discussion and reached consensuses.

## V. Discussion and Future Work

Our results show that most of the maintainability issues are found in bugs with higher severity. Thus, since these issues have a high impact on the system, it is important to resolve these issues as quickly and thoroughly as possible. Existing research[23][37] provides insightful knowledge in understanding, evaluating, and improving a system's maintainability planning, staffing, and preparation of technology for cost-effective maintenance. However, it is rarely addressed in open source software. This validates the significance of our study on connecting this knowledge to open source software.

With regard to software domain, client software contains more maintainability issues than server software. However, it also shows a decreasing trend of maintainability issues as it evolves. This result contradicts the Lehman[1] Law of declining quality. We believe that the main reason for this is due to external pressure from competitors and users. For example, Firefox as a browser is in the same market space as many other options. If the maintainability issues persist, users will be more likely to switch to a different product. Server software issues will affect all users, thus there is pressure to fix the issue as quickly as possible regardless of whether the fix will address the root causes of the bugs. When patches or temporary solutions are introduced while not directly addressing the cause of the maintainability issues, as the software evolves, the maintainability issues compound.

Currently, when software systems measure maintainability, especially with automatic code analysis approaches, they use the same metrics without considering the differences across domains. Our results highlight these differences and may indicate that it is reasonable to consider the domain classification of the software as a factor when selecting or developing metrics to measure maintainability. Moreover, the discovered decline of accessibility in client software and the rise of understandability in server software may suggest that client software systems should emphasize accessibility during earlier phases of software maintenance, while server software systems should emphasize understandability as the software matures. Furthermore, understandability is the dominant subgroup SQ in both client and server software. However, existing metrics, generally used for effort estimation and commonly associated with understandability such as cyclomatic complexity, have been found to have no or low correlation to understandability [16]. Thus, new approaches that capture facets of code understandability are in significant need.

Reopened bugs show that accessibility and understandability are also the two top SQ issues that need to be readdressed after the initial solution. Thus, suggesting software systems incorporate tools and methods to validate the solution of bugs that trigger those SQ issues to save future effort. The root causes we summarized help identify where and how to address these and other SQ issues. For example, an accessibility issue may display as an interface GUI error. However, the root cause of the issue is linked to an internal functionality error. With this knowledge, developers may verify the fix according to the root cause to confirm that the solution has addressed the root problem appropriately.

In addition to accessibility and understandability issues, portability issues have a significant impact on software systems. They tend to contain more source bugs and span across products, as well as exhibiting a trend of having more slow-fixed bugs and mostly occur in bugs with high severity. It is more difficult to improve this SQ, thus suggesting more resources should be allocated toward identifying this type of issues earlier in the life cycle.

The dependency relationships reveal a high-level connection between subgroup SQs. While most relationships were found to involve only one direct source bug, a chain of bugs can be further linked. This could help systems understand the effect of these chains, the SQ issues they bring, and how these SQ issues change within each chain of bugs, as well as provide more information for resource allocation. Thus, an extensive study is recommended to identify the common characteristics of these quality relationship chains and their impact on the rest of the software systems.

These results also represent the input for our future research agenda on the topic, mainly focusing on designing automatic ways to map bugs to SQs and also validating these findings in a larger-scale empirical study. Automated identification could be combined with automatic analysis to provide a more thorough understanding of software quality without significant additional resources.

## VI. Related Work

### A. Empirical Studies on Bug Characteristics

**Understanding general bugs**. Tan *et al.* [38] studied software bug characteristics by sampling 2,060 real world bugs in the Linux kernel, Mozilla, and Apache projects from three dimensions—root causes, impacts, and components. They also analyze the correlation between categories in different dimensions, and the trend of different types of bugs. Sahoo *et al.* [39] analyzed the software bugs based on three characteristics: observed symptoms, reproducibility, and the number of inputs needed to trigger the symptom. Their findings are mainly involved in implications for automated bug diagnosis. Ocariza *et al.* [40] [41] performed an empirical study of bug reports to understand the root causes and impact of JavaScript faults and how the results can impact JavaScript programmers, testers, and tool developers.

**Understanding specific types of bugs**. Lu *et al.* [42] presented a comprehensive study on real world concurrency bug characteristics by examining 105 randomly selected real world concurrency bugs from four representative server and client open-source applications. Their study reveals several interesting findings and provides useful guidance for concurrency bug detection, testing, and concurrent programming language design, for example, they found that around one third of the examined non-deadlock concurrency bugs are caused by violation to programmers' order intentions, which may not be easily expressed via synchronization primitives like locks and transactional memories. Li *et al.* [35] collected 709 bugs including security related and concurrency bugs. They analyzed the characteristics of those bugs in terms of root causes, impacts and software components. Their findings reveal characteristics of memory bugs, semantic bugs, security bugs, GUI bugs, and concurrency bugs. They verified their analysis results on the automatic classification results by using text classification and information retrieval techniques. Jin *et al.* [43] conducted an empirical study on performance bugs by examining 109 real world bugs. Their findings provide guidance for future work to avoid, expose, detect, and fix performance bugs.

**Understanding bugs in specific domains**. Wang *et al.* [44] studied 57 real bugs from open-source Node.js applications. They analyzed bug patterns, root causes, bug impact, bug manifestation, and fix strategies. Their results reveal findings about future concurrency bug detection, testing, and automated fixing in Node.js. Yin *et al.* [45] studied the characteristics of bugs in open source router software. They evaluated the root cause of bugs, ease of diagnosis and detectability, ease of prevention and avoidance, and their effect on network behavior. Zhou *et al.* [46] performed an empirical study on 72 Android and desktop projects. They studied how severity changes and quantified differences between classes in terms of bug-fixing attributes.

Existing empirical studies on bug characteristics focused on understanding of concurrency, performance, and security bugs without systematically incorporating quality aspects.

### *B. Empirical Studies on Maintainability*

**Ontology of maintainability**. Boehm *et al.* [22] provided an IDEF5 class hierarchy of upper-level SQs, where the top level reflected classes of stakeholder value propositions (Mission Effectiveness, Resource Utilization, Dependability, Flexibility), and the next level identified means-ends enablers of the higher-level SQs. Their studies provided the definitions, examples of the application to maintainability. Based on the ontology study, they further summarized resulting changes in the SQ ontology, and also provided examples of Maintainability need and use, quantitative relations where available, and summaries and references on improved practices [23].

**Maintainability Metrics**. Maintainability Index (MI) is the most widely used metric to quantify maintainability of any software projects. It was first introduced by Oman in 1992 [47]. The idea is to combine three code-related metrics for measuring the maintainability of a given system into a single index. Throughout the years, the original formula evolved into different versions, including Microsoft Visual Studio MI, Software Engineering Institute MI [48], and the revisited MI [4]. Heitlager [49] pointed out the shortcomings of the original MI approach, including that the result does not provide clues on what characteristics of maintainability have contributed to that value, nor on what action to take to improve this value. Sjøberg [50] stated that most of the famous software maintenance metrics including the original formula of MI are overrated and they may not reflect future maintenance effort. Senousy [51] constructed a correlation analysis of MI parameters on Linux Kernel Modules and concluded that MI was the most affected by Lines of Code, followed by Cyclomatic Complexity and lastly the Halstead Volume. Döhmen *et al.* [52] developed a benchmark aimed at determining the base rate of maintainability evolution and the properties of software that correlate with rates of maintainability evolution. Chen *et al.* [13] evaluated the software maintainability versus a set of human-evaluation factors used in the Constructive Cost Model II (COCOMO II) Software Understandability (SU) metric, through conducting a controlled experiment on humans assessing SU and performing change-request modifications on open source software (OSS) projects.

**Predicting Maintainability**. Zhou and Xu report the ability of 15 design metrics to predict how maintainable a system is, based on 148 java open source projects [53]. They found that the average control flow complexity per method to be the most important maintainability factor, and cohesion and coupling have weak impact on maintainability. Li and Henry use a combination of metrics from source code to predict the number of lines changed per class as maintenance effort of two commercial object-oriented software systems[54].

Summing up, here is how our work differs from the existing approaches:

- Instead of focusing on directly analyzing bug reports, our work focus on systematically describing software maintainability derived from bug reports and analyzing the changes of software maintainability in software evolution.
- Instead of coming up with another static analysis approach to measure software maintainability, our work applies existing high level knowledge of a maintainability ontology to bugs in practice.

## VII. Conclusion

This paper studies the characteristics of maintainability in the context of 11 products found in a large open source community from analyzing reported bugs using a maintainability ontology. We mined 61790 and manually study 6372 randomly sampled bugs and categorize them into one of the subgroup SQs found in the ontology. We further study the trend of how maintainability and its subgroup SQs change as software evolves, how related bugs are introduced in different classifications, and how thees SQs relate to each other. Overall, our results not only provide a brand new perspective on software development with a good understanding of software maintainability, but also enlighten and contribute to the current techniques of maintainability estimation and prediction. Our findings and their implications include: (1) As software evolves, products in client software exhibit an overall increasing trend of maintainability while products in server software show an opposite trend. (2) Higher severity bugs express more maintainability issues compared to lower severity, which validates the importance of software maintainability measurement. (3) The most frequent relationship originates with accessibility issues, which lead to understandability issues.

## Acknowledgement

This material is based upon work supported in part by the U.S. Department of Defense through the Systems Engineering Research Center (SERC). It is also supported by the National Science Foundation grant CMMI-1408909, Developing a Constructive Logic-Based Theory of Value-Based Systems Engineering.

## References

[1] M. M. Lehman, "Programs, life cycles, and laws of software evolution," *Proceedings of the IEEE*, vol. 68, no. 9, pp. 1060–1076, 1980.

[2] J. Koskinen, "Software maintenance fundamentals," *Encyclopedia of Software Engineering, P. Laplante, Ed., Taylor & Francis Group*, 2009.

[3] A. Ganpati, A. Kalia, and H. Singh, "A comparative study of maintainability index of open source software," *Int. J. Emerg. Technol. Adv. Eng*, vol. 2, pp. 228–230, 2012.

[4] K. D. Welker, "The software maintainability index revisited," *CrossTalk*, vol. 14, pp. 18–21, 2001.

[5] P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice." *IEEE software*, vol. 29, no. 6, 2012.

[6] W. Cunningham, "The wycash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, pp. 29–30, 1993.

[7] D. I. Sjøberg, A. Yamashita, B. C. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Transactions on Software Engineering*, vol. 39, no. 8, pp. 1144–1156, 2013.

[8] A. Yamashita, "Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data," *Empirical Software Engineering*, vol. 19, no. 4, pp. 1111–1143, 2014.

[9] A. Yamashita and L. Moonen, "Do code smells reflect important maintainability aspects?" in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 306–315.

[10] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[11] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer, "Managerial use of metrics for object-oriented software: An exploratory analysis," *IEEE Transactions on software Engineering*, vol. 24, no. 8, pp. 629–639, 1998.

[12] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng, "Decoupling level: a new metric for architectural maintenance complexity," in *Proceedings of the 38th International Conference on Software Engineering*. ACM, 2016, pp. 499–510.

[13] C. Chen, R. Alfayez, K. Srisopha, L. Shi, and B. Boehm, "Evaluating human-assessed software maintainability metrics," in *Software Engineering and Methodology for Emerging Domains*. Springer, 2016, pp. 120–132.

[14] B. W. Boehm, R. Madachy, B. Steece *et al.*, *Software cost estimation with Cocomo II with Cdrom*. Prentice Hall PTR, 2000.

[15] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE Computer Society, 2009, pp. 367–377.

[16] S. Scalabrino, G. Bavota, C. Vendome, M. Linaresvasquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability: How far are we?" in *Ieee/acm International Conference on Automated Software Engineering*, 2017, pp. 417–427.

[17] H. Wu, L. Shi, C. Chen, Q. Wang, and B. Boehm, "Maintenance effort estimation for open source software: A systematic literature review," in *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 2016, pp. 32–43.

[18] A. Adewumi, S. Misra, N. Omoregbe, B. Crawford, and R. Soto, "A systematic literature review of open source software quality assessment models," *SpringerPlus*, vol. 5, no. 1, p. 1936, 2016.

[19] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[20] R. P. de Oliveira and E. S. de Almeida, "Evaluating lehman's laws of software evolution for software product lines," *IEEE Software*, vol. 33, no. 3, pp. 90–93, 2016.

[21] L. Yu and A. Mishra, "An empirical study of lehman's law on software quality evolution." *Int. J. Software and Informatics*, vol. 7, no. 3, pp. 469–481, 2013.

[22] B. Boehm and N. Kukreja, "An initial ontology for system qualities," *Incose International Symposium*, vol. 25, no. 1, p. 341356, 2015.

[23] B. Boehm, C. Chen, K. Srisopha, and L. Shi, "The key roles of maintainability in an ontology for system qualities," *Incose International Symposium*, vol. 26, no. 1, pp. 2026–2040, 2016.

[24] E. Fricke and A. P. Schulz, "Design for changeability (dfc): Principles to enable changes in systems throughout their entire lifecycle," *Systems Engineering*, vol. 8, no. 4, pp. 308.1–308.2, 2005.

[25] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, "Improving software diagnosability via log enhancement," in *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 3–14.

[26] A. Kavcic, "Software accessibility: Recommendations and guidelines," in *The International Conference on Computer As A Tool*, 2006, pp. 1024–1027.

[27] B. S. Blanchard, D. Verma, and E. L. Peterson, *Maintainability: a key to effective serviceability and maintenance management*. John Wiley & Sons, 1995, vol. 13.

[28] D. Bjørner, "Facets of software development," *Journal of Computer Science and Technology*, vol. 4, no. 3, pp. 193–203, 1989.

[29] E. R. Poort, N. Martens, V. D. W. Inge, and H. Van Vliet, "How architects see non-functional requirements: beware of modifiability," in *International Conference on Requirements Engineering: Foundation for Software Quality*, 2012, pp. 37–51.

[30] R. Sanchez and J. T. Mahoney, "Modularity, flexibility, and knowledge management in product and organization design," *Strategic Management Journal*, vol. 17, no. S2, pp. 63–76, 2015.

[31] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and scalability of ejb applications," in *Proc. 2002 ACM Sigplan Conference on Object-Oriented Programming Systems, Languages and Applications*, 2002, pp. 246–261.

[32] B. Boehm and H. In, "Identifying quality-requirement conflicts," in *International Conference on Requirements Engineering*, 1996, p. 218.

[33] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, 2012.

[34] M. Leszak, D. E. Perry, and D. Stoll, "Classification and evaluation of defects in a project retrospective," *Journal of Systems and Software*, vol. 61, no. 3, pp. 173–187, 2002.

[35] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?:an empirical study of bug characteristics in modern open source software," in *The Workshop on Architectural and System Support for Improving Software Dependability*, 2006, pp. 25–33.

[36] M. Coltheart, "The semantic error: Types and theories," *Deep dyslexia*, pp. 146–159, 1980.

[37] A. L. Rector, "Normalisation of ontology implementations: Towards modularity, re-use, and maintainability," in *EKAW Workshop on Ontologies for Multiagent Systems*. Citeseer, 2002.

[38] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[39] S. K. Sahoo, J. Criswell, and V. Adve, "An empirical study of reported bugs in server software with implications for automated bug diagnosis," in *ACM/IEEE International Conference on Software Engineering*, 2011, pp. 485–494.

[40] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side javascript bugs," in *ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 55–64.

[41] F. S. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "A study of causes and consequences of client-side javascript bugs," *IEEE Transactions on Software Engineering*, vol. 43, no. 2, pp. 128–144, 2017.

[42] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 2, pp. 329–339, 2008.

[43] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, 2012, pp. 77–87.

[44] J. Wang, W. Dou, Y. Gao, C. Gao, F. Qin, K. Yin, and J. Wei, "A comprehensive study on real world concurrency bugs in node.js," in *Ieee/acm International Conference on Automated Software Engineering*, 2017, pp. 520–531.

[45] Z. Yin, M. Caesar, and Y. Zhou, "Towards understanding bugs in open source router software," *Acm Sigcomm Computer Communication Review*, vol. 40, no. 3, pp. 34–40, 2010.

[46] B. Zhou, I. Neamtiu, and R. Gupta, "Experience report: How do bug characteristics differ across severity classes: A multi-platform study," in *IEEE International Symposium on Software Reliability Engineering*, 2016, pp. 507–517.

[47] P. Oman and J. Hagemeister, "Metrics for assessing a software systems maintainability," in *Software Maintenance, 1992. Proceerdings., Conference on*. IEEE, 1992, pp. 337–344.

[48] E. VanDoren, "Maintainability index technique for measuring program maintainability. software engineering institute," 2002.

[49] I. Heitlager, T. Kuipers, and J. Visser, "A practical model for measuring maintainability," in *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*. IEEE, 2007, pp. 30–39.

[50] D. I. Sjøberg, B. Anda, and A. Mockus, "Questioning software maintenance metrics: a comparative case study," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2012, pp. 107–110.

[51] M. B. Senousy and T. S. Mazen, "Correlations and weights of maintainability index (mi) of open source linux kernel modules," *International Journal of Computer Applications*, vol. 91, no. 7, 2014.

[52] D. C. J. V. Till Döhmen, Magiel Bruntink, "Towards a benchmark for the maintainability evolution of industrial software systems," in *2016 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA)*. IEEE, 2016, pp. 11–21.

[53] Y. Zhou and B. Xu, "Predicting the maintainability of open source software using design metrics," *Wuhan University Journal of Natural Sciences*, vol. 13, no. 1, pp. 14–20, 2008.

[54] W. Li and S. Henry, "Object-oriented metrics that predict maintainability," *Journal of systems and software*, vol. 23, no. 2, pp. 111–122, 1993.