Valentina Piantadosi

On the Evolution of the Code Readability

PhD Thesis

University of Molise, Italy

May, 2022

University of Molise

Department of Biosciences and Territory

Ph.D. Course in Biosciences and Territory
XXXIV Cycle
S.S.D. ING-INF/05

Ph.D. Thesis

On the Evolution of the Code Readability

Ph.D. Coordinator
Prof. Giovanni Fabbrocino

Tutor
Prof. Rocco Oliveto

Ph.D. Candidate
Valentina Piantadosi

Academic Year 2020/2021

Abstract

Code reading is an activity frequently performed by developers. Before modifying code, developers have to read it, especially if it was authored by others. Several studies have been carried out to find insights related to code readability. However, they were mostly conducted on single (static) code snippets, *i.e.,* they did not take into account the ever-changing nature of software. In a preliminary study, we observe that refactoring operations performed by developers to improve several non-functional aspects of open-source software systems often result in improved code readability.

Motivated by such results, in this thesis, we studied code readability in the context of software evolution. Especially, we aimed at understanding: (i) to what extent software developers are interested in code readability; (ii) how code readability evolves in complex software systems; (iii) the influence of developers' personal characteristics on the evolution of code readability.

First, we observed that the large majority of the developers we surveyed reported that they consider readability as an important aspect of the source code. Then, by mining several software repositories we observed that readability rarely changes in software evolution. Therefore, files created unreadable rarely become readable, and vice versa. Finally, we noticed that some personal characteristics of developers have an impact on code readability. In particular, developers' orienting network (related to their attention) positively correlates with code readability.

Contents

II Code Readability Improvement & Evolution 27

III Code Readability & Cognitive Human Factors 125

List of Figures

List of Tables

CHAPTER 1

Introduction

Code reading is one of the most frequent activities performed by developers. Developers have to read (and comprehend) the code, before modifying it. Erlikh [86] demonstrated that developers spend more time in their maintenance activities than to write code from scratch. Given the importance of code readability, researchers have investigated the factors that could influence code readability [146, 164, 48]. Subsequently, these factors have been used to construct new models to automatically assess code readability [59, 181, 80, 200, 199], in particular, and code quality, in general.

In a preliminary study conducted to understand what the real motivations that push developers to perform refactoring operations are [171], we observed that refactoring operations help developers to improve, among other aspects, code readability. Also, through the manual investigation of 551 pull requests, we observed that 468 refactoring operations (∼85%) have been specifically performed to improve code readability.

The results achieved in such a study motivate this thesis. Especially, all the studies carried out to study code readability have been conducted considering single and, more important, static code snippets without considering the natural and continuous evolution of software. Thus, the *goal* of this thesis is to study the evolution of code readability during the life-cycle of a software system. As a *first contribution*, we aim at understanding to what extent code readability is important to developers during the evolution and the maintenance of a software system. To this aim, we interviewed 122 open-source developers. The results achieved show that ∼83.8% of developers consider code readability as an important factor in their source code writing activities. This study complements and corroborates the findings of our preliminary study [175].

As a *second contribution*, we tried to understand how readability evolves in software systems. Specifically, we conducted a mining study on 25 open-source software systems and ∼83,000 commits to analyze the readability evolution during daily software development activities (*i.e.,* creation, modification and deletion). To do this, we created a model to represent the readability evolution of a file in a software project. Results show that code readability of a file can undergo small changes during the development of a software system that does not result in a change of code from *readable* to *unreadable* or *vice versa*. In particular, if a code component is created unreadable, it will likely remain unreadable during the entire software system evolution process [175].

Finally, given the human nature of code writing, as a *third contribution*, we empirically verify if cognitive human aspects can influence the readability of a code snippet. Especially, we conducted an empirical study in which we invited 32 software developers [177] to participate in a controlled experiment. We profiled participants with 3 psychometric tests to measure attention level (divided into three networks: *alerting*, *orienting* and *executive control*), working memory, and immediate recall (aspect related to episodic memory). In the context of the experiment, developers were asked to carry out code writing tasks. The achieved results show that one of the attention-related factors, *i.e.,* the orienting network, correlates with code readability (developers with a higher orienting produce more readable code).

The remainder of this thesis is organized as follows. In Part I we provide background notions and a literature review on the main topics of the thesis: Refactoring (Chapter 2), code readability and understandability (Chapter 3), and cognitive human aspects (Chapter 4). In Part II we present the studies conducted on the evolution of code readability. Especially, in Chapter 5 we report the study that motivates this thesis, *i.e.,* the study aiming at assessing the motivations behind refactoring operations [171]. In Chapter 6 we report the study assessing the importance of code readability for developers during maintenance tasks [175]. This second part concludes with Chapter 7, in which we report the mining study aimed at assessing the evolution of code readability in open-source projects [177]. In Part III we report the empirical study conducted to analyze whether cognitive human aspects can influence code readability. Finally, in Chapter 9 we conclude the thesis by summarizing the main findings achieved.

Part I

Background & Related Works

*“Knowledge is power.
Knowledge shared is power multiplied.”*

Robert Boyce

CHAPTER 2

Software Refactoring

Contents

When a developer performs a simple change on the code, it loses its structure. Over time so many changes deteriorate the code structure. Thus, a developer has to tide up the code through an activity called refactoring [96]. *Refactoring* is the activity performed by the developer for changing a software system. Specifically, this activity improves the internal structure of software without changing its external behaviour. With this activity, developers can clean up code minimizing the introduction of bugs in the code [96].

In Software Engineering research, there are many studies that analyze refactoring operations from different perspectives, including how developers perform refactoring [158]; the relationship between refactoring and other software-related activities (*e.g.,* merge conflicts [140]); the impact of refactoring operations on the likelihood of introducing bugs [46]; the impact of refactoring on specific quality indicators (*e.g.,* quality metrics) [215, 216, 33, 66] or on developers' productivity [152].

2.1 Refactoring Practices

Murphy *et al.* [156] created Mylar Monitor, a framework to collect and store trace information of a user's activity in Eclipse. This framework collects all the events performed by users (*e.g.,* changes, window events, URLs viewed through the embedded Eclipse browser) including the use of the refactoring operations provided by the IDE. From the analysis of collected data, the authors found evidence of the developers using 11 kinds of refactoring commands and the most common operation is the "Rename refactoring".

Murphy-Hill *et al.* [158] investigated how developers perform refactorings. The authors analyzed refactoring operations in eight different datasets. For example, one of the studied datasets contains usage data from 41 developers using the Eclipse development environment, capturing an average of 66 hours of activity per developer. Some of the several interesting findings are that (i) programmers rarely configure refactoring tools (the overall mean change frequency varying between 10% and 12%); (ii) commit messages do not help in predicting refactoring, since developers do not explicitly report their refactoring activities in them; (iii) developers often interleave refactoring with other programming activities; and (iv) most of the refactoring operations (~90%) are performed without the help of any tools.

2.2 Motivations and Actions behind Refactoring Operations

Wang *et al.* [231] interviewed developers from four software companies to reveal the major factors motivating refactoring operations. Results highlight external motivators, *e.g., Recognitions from Others*, and intrinsic motivators, *i.e.,* when refactoring is initiated without any obvious external reward (*e.g., Self Esteem*) behind refactorings.

Kim *et al.* [119] presented a field study surveying and interviewing 328 Microsoft engineers to investigate when and how engineers refactor code. Subsequently, the authors checked if there was a match between survey respondents' perception and reality. From this study, it is possible to see that the refactoring definition is not completely applicable. The execution of a refactoring operation can have costs and risks, but it can ask also for various types of tool support. They identified the low readability of source code as one of the symptoms that push developers to perform refactoring (mentioned by 21% of developers).

Silva *et al.* [208] observed that the previously discussed works [231, 119] report findings from surveys asking developers about their general refactoring habits without focusing on real refactorings they performed. Thus, Silva *et al.* [208] used RMiner [222] to monitor refactorings performed in open source repositories and contacted the developers that perform the refactorings asking them to motivate the performed changes. Then, they grouped the responses and defined a catalogue of 44 motivations for 12 refactoring operations. For example, they find that the *Rename Package* refactoring is executed to improve the package name, to enforce naming consistency, and to move the package to the appropriate container [208].

Peruma *et al.* [174] also analyzed the motivations pushing developers to refactor their code, but they focused only on rename refactoring operations. These operations are one of the main activities of a developer to maintain and evolve the software. Developers rename an entity (*e.g.,* variable or method) when their labels do not correspond to their behaviour. For example, if the software has a big number of changes, the behaviour of an entity can change and the entity name change consequently. Thus, the authors performed an empirical study on motivations based on rename refactorings. The principal motivation obtained from their study is that a developer renames an entity for adapting its name to its behaviour.

Finally, Vassallo *et al.* [229] mined 200 systems to quantitatively investigate factors correlating with refactoring. Specifically, they analyzed (i) what are types of refactoring applied, (ii) when are refactoring operations performed, and (iii) what factors induced developers to do refactoring. They found that (i) most refactorings are performed while enhancing existing features, (ii) refactorings are mostly performed after one year since the project started and rarely before a new release, and (iii) developers that refactor code are often the owners of impacted files.

2.3 The Impact of Refactoring on Quality Metrics

In the existing literature, there no studies on the analysis of the relations between metrics/code smells and refactoring activities' developers. Bavota *et al.* [44] studied this relationship. The authors extracted the history of three Java open source projects. From these projects, the authors analyzed if the developer executes refactoring operations on a given code component, when the quality metrics indicate the need to refactor it or if there is a code smell. Results show that there is no relationship between quality metrics and refactoring operations: Developers refactor code components, also if these are not indicated from quality metrics.

Chávez *et al.* [66] presented a study of how refactoring operations impact on five internal quality attributes, *i.e.,* cohesion, coupling, complexity, inheritance, and size. The authors studied the history of 23 open source systems, composed of 29,303 refactoring operations. Results show that developers perform these activities to improve internal quality attributes. Effectively, the code quality improves after these operations, also when developers perform these operations to add new functionality or to fix a bug.

AlOmar *et al.* [31] identified what fefactoring operations are performed to improve quality metrics to optimize the quality. They mined (i) 3,759 curated open-source Java projects to extract all design-related refactoring activities applied and documented by developers and structural metrics, and (ii) 1,245 quality improvement commits with their corresponding refactoring operations. Results show that structural metrics are more popular to represent internal quality attributes (*e.g.,* CBO, LOC and WMC) and developers consider some quality attributes than others.

Almogahed *et al.* [30] highlighted a known problem: Developers do not know what is the more appropriate refactoring technique to use. For this reason, they searched studies that categorize refactoring techniques on the influence of quality attributes. Unfortunately, in literature, there is no categorization of refactoring techniques considering the influence of quality attributes. Thus, developers cannot resolve these issues, but authors provide recommendations to researchers for filling this gap. Researchers could create reference and prediction models and developers might use them as guidelines in the selection of apposite refactoring techniques.

2.4 Refactoring and Other Software-Related Activities

Some works have studied the relationship between refactoring and other software-related activities or properties. Stroggylos and Spinellis [215] extracted refactoring operations from version control system logs of three open-source libraries to study the impact of refactoring operations on the values of nine object-oriented quality metrics. Their results show how some quality metrics influence negatively refactoring.

Moser *et al.* [152] conducted a case study in a close-to industrial environment investigating the impact of refactoring on the productivity of an agile team and the produced code quality. In the context of mobile apps development, the achieved results show that refactoring increases software quality and developers' productivity. Cedrim *et al.* [64] presented a similar study. Their findings, however, indicate that in most cases code quality does not improve after refactoring operations.

Alshayeb [33] also investigated the link between refactoring operations and software quality. Particularly, the author studied the impact of refactoring operations on five external quality attributes (adaptability, maintainability, understandability, reusability, and testability). His findings highlight that benefits brought by refactoring operations on some code classes are often counterbalanced by a decrease of quality in other classes.

Bavota *et al.* [43] studied the extent to which refactoring activities induce faults. The authors showed that refactorings involving hierarchies (*e.g., push down method*) that can induce faults more frequently than others that are likely to be harmless in practice.

Szoke *et al.* [216] performed a case study on five large-scale systems to investigate the relationship between refactoring and code quality. The authors investigated 2 million lines of code and they found 200 commits related to refactoring activities. Their findings show that small refactoring operations performed in isolation rarely impact software quality. On the other side, a high number of refactoring operations performed in a block can result in a notable code quality improvement.

Mahmoudi *et al.* [140] presented a large scale study on ∼3000 open-source Java repositories investigating the link between merge conflicts in multiple development branches and code refactoring operations. The authors found that refactoring operations are involved in 22% of merge conflicts. Also, conflicts involving refactoring operations tend to be more complex to solve as compared to conflicts caused by other types of changes.

Finally, Lin *et al.* [138] studied the relationship between refactoring operations and their effect on source code "naturalness". Naturalness of a software is a property to evaluate to what extent the code surprises the developer [111]. Their results show that refactorings do not necessarily increase the naturalness of code and its impact depends heavily on the type of refactoring.

CHAPTER 3

Source Code Readability

Contents

Quality of the source code has been extensively studied in the literature starting from the 1960s when first software metrics have been proposed, like *Lines Of Code*, *McCabe's Cyclomatic Complexity* [148] and *Halstead's metrics* [107].

While none of them explicitly refers to readability, many of them include related notions of maintainability or understandability [148, 54, 101]. ISO 9126 *understandability* takes the perspective of the end-user rather than one of the developers as, for instance, in the case of the earlier model by Boehm *et al.* [54] and SQALE [134]. Given the extensive research on software quality, providing a complete overview is not possible and we focus on readability from here on.

3.1 Code Readability

Previous works focused on automatic assessment of code readability [58, 59, 181, 80, 199]. All the state-of-the-art approaches use machine learning: they all define some features measured on the source code and they train a binary classifier to distinguish *readable* code from *unreadable* code. To train the classifier on how to correctly classify source code snippets, these approaches need a huge dataset containing human assessments of code readability. Three datasets are available in the literature [58, 80, 200] and these are built similarly: the authors selected a set of snippets S, asked several developers to evaluate them in terms of readability using a Likert scale from 1 to 5 and then they (i) aggregated such values and (ii) used a threshold to have a ground-truth readability level to classify each snippet as *readable* or *unreadable*. In all these studies, authors reported the agreement among the evaluators.

The study of Buse *et al.* [58] differs from others [80, 200] because the authors extracted structural features, *e.g.,* line length, identifier length, number of loops, and so on. The authors obtained a positive correlation with software quality metrics, code changes and defect reports. Thus, Buse *et al.* [58, 59] designed the first readability model based on structural features. Instead, the study of Dorn [80] extended the model created by the study of Posnett *et al.* [181]. The model of Posnett *et al.* [181] used Halstead's metrics and it was used to estimate the readability of text in natural language, considering three specific features: lines of code (LOC), entropy and volume. Dorn [80] introduced visual, spatial and linguistic features to the previous models that measure aspects such as indentation regularity and alignment, both important when reading code. He showed that such features allow to define a more general model [80].

Considering the previous studies [58, 80], Scalabrino *et al.* [200, 199] defined a new set of textual features to capture a different dimension of code readability. Such features include, for example, consistency between comments and identifiers, and comment readability. The authors showed that a comprehensive model including all the state-of-the-art features is more accurate than single models that consider only single categories of features.

In literature, some studies have discussed these metrics. The first is the study of Pantiuchina *et al.* [170], in which the authors tried to understand if such readability metrics change in the commits in which developers declared they improved code readability. Their results show that this happens only in a minority of the cases. Also, Fakhoury *et al.* [88] recently reported a similar result. It is possible to obtain this result because even if a change improves readability, the improvement might be small concerning the size of the changed class, and so this makes the difference in the metric It is possible to obtain this result because even if a change improves readability, the improvement might be small concerning the size of the changed class, and so this makes explainable the difference in the metric explainable.

Lee *et al.* [132] observed that the number of coding violations increases during the early stages of the project history (planning, pre-alpha, alpha), but it drops at the beta level. They generally showed a decreasing trend of coding violations as the project matures. Also, they showed that changes in code readability are related only to some types of coding violations. Using several readability indicators, Spinnellis *et al.* [214] observed that the readability of Unix gradually improved. However, they also highlighted that there is insufficient evidence to claim that readability is still increasing.

In literature, there are different code readability metrics. Each code readability metric uses different aspects of source code (*i.e.,* structural features [58, 59]; visual, spatial, and linguistic features [80]; textual features [200, 199]). There are studies to understand if developers changed code readability after their declarations [171, 88]. Despite these studies, in literature, there is no study to analyze the evolution of code readability in open-source software. In addition, there is limited empirical evidence on the importance of code readability for developers. We bridge this gap with the study in Chapter 6. Finally, no one has tried to provide guidelines to developers to avoid code readability erosion. We bridge this gap with the study in Chapter 7.

3.2 Software/Code Understandability

Code readability represents how easily information is conveyed to the developer. Several works focused, instead, on measuring a related concept, *i.e.,* code understandability or comprehensibility, which represents to what extent the information in the code is usable by the developer.

Capiluppi *et al.* [62] defined a measure of understandability in OSS projects. The goal of this study is to understand how understandability evolves. Specifically, this measure is computed on the history of 19 open source projects considering (i) the percentage of micro-modules located in the macro-modules (*i.e.,* the number of files within the directories), and (ii) the relative size of the micro-modules. Their results demonstrate that, during the lifecycle of the system, the understandability typically grows.

Misra *et al.* [150] compared the Weyuker's properties [235] and other similar measures with the Cognitive Weigth Complexity Measure (CWCM). A Cognitive Weight Complexity Measure (CWCM) is given from the cognitive weights of basic control structures, $CWCM = W_c$ in which W_c is the cognitive weight of the basic control structure, that is a basic building block of software. Specifically, W_c is the sum of the cognitive weight of all linear blocks represented by basic control structures (BCS). Thus, Misra *et al.* [150] assigned weight to software components through the analysis of their control structures. Instead, Weyuker's properties [235] are focused on nine properties to measure software complexity in terms of weaknesses of a proposed measure.

Thongmak *et al.* [218] evaluated the understandability of aspect-oriented software through aspect-oriented software dependence graphs. To do this, the authors proposed metrics that can operate on different levels of dependence graphs (*i.e.,* Module-Level metrics, Class/Aspect-Level metrics, System-Level metrics). Module-Level metrics have to operate for understanding (i) *methods* based on parameters, statements, called methods, and formal and actual parameters of each statement; (ii) *advice* based on parameters, statements, called methods, parameters-in/out of each statement, and pointcuts. Class/Aspect-Level Metrics consider the relationships between modules (*i.e.,* class membership dependencies and parameter dependencies). System-level metrics consider graphs representing the entire system.

Subsequently, Chen *et al.* [67] explored the COCOMO II Software Understandability metrics verifying a correlation between the software maintainability and a set of human-evaluation factors. COCOMO II is the acronym of COnstructive COst MOdel II and it is a model to measure software effort, cost, and schedule estimation [55]. COCOMO II is based on three submodels, *i.e.,* the early design model, the application composition model, and the post-architecture model. This last submodel is used to predict software development and maintenance effort. With the use of some factors, it is possible to calculate the cost behind a change on the existing software. In these factors, it is possible to see that there are SU factors (structure, application clarity, self-descriptiveness), used for evaluating the maintenance and adaptation underestimates [55]. Chen *et al.* [67] conducted a controlled experiment with a study involving six graduate students, who were asked to accomplish 44 maintenance tasks. This study shows that higher quality of the code structure and higher self-descriptiveness lead to higher code quality [67].

Scalabrino *et al.* [197, 198] performed an in-depth analysis of 121 metrics divided into three categories: (i) *new code-related metrics*, (ii) *documentation-related metrics*, and (iii) *developer-related metrics.* In **code-related metrics**, authors consider the following set of metrics:

- *Kasto and Whalley's metrics* [117]: *cyclomatic complexity* [148] (*i.e.,* the number of linear independent paths of the snippet), the *average number of nested blocks*, *number of parameters*, *number of statements*, and *number of identifiers.*

- *Buse and Weimer's metrics* [59], which are metrics for measuring properties at the level of a single line of code in aggregate form with the mean and/or the maximum. Specifically, Buse and Weimer measure: (i) the *average* of the number of assignments, number of blank lines, number of commas, number of comments, number of comparisons, number of conditionals, number of loops, number of operators, number of parentheses, number of periods, and number of spaces; (ii) *average and maximum* of number of identifiers, number of keywords and numbers, identifiers length, indentation length, and line length; (iii) *maximum* of number of words.

- *Posnett et al.'s metrics* [181]: *lines of code* (LOC), *token entropy*, and *Halstead's volume.*

- *Dorn's metrics* [80], which are a different version of Buse and Weimer's metrics [59] because Dorn measures the bandwidth of the Discrete Fourier Transform (DFT) of most previous metrics. Indeed, Dorn [80] measures the DFT on all the properties listed from the *Buse and Weimer's metrics*, except the number of blank lines, the number of characters, the number of literals, the number of strings, and the length of the identifiers. Furthermore, Dorn [80] measures (i) the absolute and the relative area (*i.e.,* total area of comments and area of comments separated by an area of strings) of characters on different token categories (*i.e.,* comments, identifiers, keywords, literals, numbers, operators, and strings), (ii) the number of aligned blocks, and (iii) the number of identifiers that include words in the English dictionary.

- *Scalabrino's metrics* [200], which are metrics to evaluate code readability calculating the average, the maximum and the minimum, *i.e., Narrow Meaning Identifiers* (NMI), *Identifiers Terms In Dictionary* (ITID), *Textual Coherence* (TC) and *Number of Meanings* (NM), *i.e.,* the metric defined in this same study [200].

- *Invoked Methods Signature Quality* (IMSQ), a new code-related metric defined in the study of Scalabrino *et al.* [198]. This metric measures the readability and the representativeness of the signature of internal methods invoked by given code snippets belonging to the same system considering the minimum, the average and the maximum values.

In **documentation-related metrics**, Scalabrino *et al.* [197, 198] use the following metrics to capture the quality of the internal documentation of a snippet:

- *Scalabrino's metrics* [200]: (i) *Comments Readability* (CR), which measures readability values of the comments in a snippet with the use of Flesch reading-ease test [93]; (ii) *Comments and Identifiers Consistency* (CIC), which measures the consistency between comments and code; (iii) CIC_{syn}, which measures the consistency between comments and code considering synonyms.

- new documentation-related metrics [198]: these two metrics evaluate respectively the internal documentation quality, *i.e., Methods Internal Documentation Quality* (MIDQ), and the external documentation quality, *i.e., API External Documentation Quality* (AEDQ).

In the end, in **developer-related metrics**, Scalabrino *et al.* [197, 198] capture:

- the programming experience of the developer in years (PE_{gen});
- the programming experience in years of a developer of the programming language related to the implemented snippet (PE_{spec});
- *External API Popularity* (EAP).

Scalabrino *et al.* [197, 198] correlated such metrics — singularly and combining them — with several proxies for code understandability. Using a dataset of 444 human evaluations from 63 developers, the authors showed that none of such metrics is related to understandability, including code readability. Results of interviews of five professional developers suggest that code readability is important to them whatsoever. The authors conclude that readability may affect understandability in the long run, *i.e.,* unreadable code may tire developers more quickly.

Trockman *et al.* [220] performed a reanalysis of the dataset by Scalabrino *et al.* [197] through the combination of metrics and statistical modelling techniques. They showed that the combination of metrics improves the assessment of code understandability, as also reported by Scalabrino *et al.* [198].

CHAPTER 4

Cognitive Human Factors

Contents

In this chapter, we first provide some background on psychological notions, such as cognitive functions and aspects. Then, we discuss both (i) related work exploring generic applications of the cognitive aspects (*e.g.,* driving and mathematics), and (ii) previous applications of cognitive psychology in the Software Engineering domain.

4.1 Psychological Notions

In cognitive psychology, the term *cognition* is composed by from thoughts and ideas and it is used to denote the internal mental processes (or *cognitive* *functions*). Cognitive functions regulate human perception, reasoning, memory, intuition, thinking, speaking, decision making, and problem solving [195, 49]. Such processes cannot be observed directly but they can be measured indirectly through psychometrics, by using specific tests. Such tests may be, for example, questionnaires or exercises, and they aim at measuring proxy metrics, such as reaction times, psychological responses, or real-time neuroimaging [99, 29].

Cognitive functions can be measured through psychometrics tests and neuroimaging. For each cognitive function, there is an appropriate psychometric test. For example, to measure the working memory, it is possible to use the Symbol Digit Modalities Test and the List Sorting Working Memory Test [100]. The Brief Test of Adult Cognition by Telephone (BTACT) can be used to measure different aspects of the cognitive functions (*e.g.,* immediate recall) [226]. For measuring attention level, it is possible measure through the Attention Network Test (ANT) [179, 89, 230]. The proxy variables measured after or during the tests may then be combined to provide a final measure of an *aspect* of the cognitive function (what we call, in this study, *cognitive human aspect*). In the case of attention level, we can measure three different networks, *i.e., alerting*, *orienting*, and *executive control.* Specifically, *alerting* is the ability to achieve and maintain an alert state, *orienting* is the ability to select the information from sensory input, and *executive control* is the ability to understand how resolve conflict among responses. To measure these networks, Fan *et al.* [89] designed the Attention Network Test (ANT). This test is composed of four cue conditions (*i.e.,* no cue, center cue, double cue and spatial cue) and six stimuli (*i.e.,* neutral, congruent and incongruent). In ANT, reaction times obtained in different situations are combined to measure the *efficiency* of the *alerting*, *orienting*, and *executive control* cognitive functions. *Alerting* is the subtraction between the mean RT of the double cue conditions and the mean RT of the no cue conditions. *Orienting* is the subtraction between the mean RT of the spatial cue conditions from the mean RT of the center cue. *Executive control* is the subtraction between the mean RT of all congruent stimuli and the mean RT of all incongruent stimuli. A different approach used for studying cognitive functions is through *cognitive neuroscience.* Functional neuroimaging is used to find correlation of the patterns obtained from the brain activity in various processing stages and different cognitive functions [195].

4.2 Relation between cognitive aspects and task performance

Previous work focused on studying cognitive functions in different domains. In some experiments, researchers even tried to explain or predict the outcome of non-trivial tasks (*e.g.,* driving). In these studies, the authors investigate the relation between cognitive aspects and task performance of any nature (*i.e.,* linguistic, mathematical and driving).

In **linguistic research**, Nour *et al.* [160] analyzed the attention network tests in three different groups of participants (*i.e.,* intepreting students, translation students, and professional interpreters. Results show the correlation between two types of language interpretations (*i.e.,* professionals and students) and the attention networks, *i.e., alerting*, *orienting* and *executive control.* Results shown that for each group of participants has a specific attention network dynamics: for example, interpreting students differ from translation students for alertness and executive network. Woumans *et al.* [239] studied the relation between language control and non-verbal cognitive control between monolingual, Dutch-French unbalanced bilinguals, balanced bilinguals, and interpreters. Results show that bilinguals can modulate better the nature and extent of a cognitive control advantage compared to interpreters and monolinguals.

Multiple studies demonstrated that the experience in a certain domain can allow the acquisitions of skills for other domains with similar abilities [103, 52, 104, 202] and that a training of some cognitive functions can change attentional processes [137].

Other studies used cognitive human aspects in the **mathematical field** to evaluate whether there is a correlation between cognitive human aspects and elementary mathematics performance [105, 155, 172]. Passolunghi *et al.* [172] found precursors of mathematics learning in the primary school. Authors verified the correlation between cognitive abilities (*i.e.,* working memory and counting ability) and mathematics learning. Their results show that cognitive abilities are associated with mathematics learning. Wei *et al.* [234] studied which cognitive human aspects are necessary to obtain advanced mathematics knowledge and skills. Results show that spatial abilities and language comprehension have a strong correlation with performance in advanced mathematics, but the study does not report any significant correlation with numerical processing. Musso *et al.* [159] also considered interactions and influence of cognitive human aspects on mathematical performances. Results show a relevance on educational quality, improvement, and accountability.

Weaver *et al.* [233] used *alerting*, *orienting* and *executive control* efficiency to predict the driving outcome. For this reason, Weaver *et al.* [233] wanted to verify whether the Useful Field of View[1] is equivalent to the Attention Network Test (ANT). Results have confirmed this equivalcence.

Multiple studies demonstrated that the experience in a certain domain can allow the acquisitions of skills for other domains with similar abilities [103, 52, 104, 202] and that a training of some cognitive functions can change attentional processes [137].

4.3 Cognitive Human Aspects in Software Engineering

In *Software Engineering research*, previous work measured cognitive human aspects and correlated them with developers' characteristics and software quality (mostly related to the API usability).

A line of research support our theory. In last decade, neuroimaging techniques have been used to understand the cognitive processes of programming [84, 94, 112, 173, 206, 207, 126, 116]. Specifically, Peitek *et al.* [173] and Siegmund *et al.* [207] used these techniques in the program comprehension and Krueger *et al.* [126] used these in the code writing.

Peitek *et al.* [173] analyzed whether functional magnetic resonance imaging (fMRI) can measure program comprehension. The authors invited 17 developers

[1] The Useful Field of View is used to predict driving performances and it is used to measure the processing speed.

to comprehend source code in an fMRI (functional magnetic resonance imaging) scanner. Results show that five brain regions are activated during a program comprehension task related to working memory, attention level and language processing. The cognitive effort is reduced given developers' familiarity with the programming language. Subsequently, this experiment has been replicated with 11 participants and results have been confirmed by Siegmund *et al.* [207]. [207] used the fMRI with 11 participants while they read a program. Authors perform manipulations on experimental conditions of beacons and layout to understand cognitive processes of the bottom-up comprehension. Their results show that beacons facilitate program comprehension tasks and there is less brain activation.

Another study that involves functional magnetic resonance imaging (fMRI) is the study of Krueger *et al.* [126]. Authors compare neural representations of code writing and those of prose writing. Results show the the pose writing active left hemisphere that are associated with language and the code writing involves the activation of more parts of the right hemisphere, *i.e.,* attention control, working memory, planning and spatial cognition. Thus, these results support the evidence that code and prose writing have different behaviour at mental level.

Sharafi *et al.* [203] perform two controlled experiments with 112 students during a series of development activities, *i.e.,* code comprehension, code review, and data structure manipulations. Students was analyzed through neuroimaging activities, *i.e.,* functional near-infrared spectroscopy (fNIRS) and functional magnetic resonance imaging (fMRI) and eye tracking. Results show that there are different neural representations between programming languages and natural languages.

As described in Section 4.1, cognitive human aspects can be measured through psychometric tests and neuroimaging. In software engineering, the functional neuroimaging take at light that given development activities correspond to the neural activation. This has been demonstrated by Peitek *et al.* [173], Siegmund *et al.* [207] and Krueger *et al.* [126]. These studies shows that programming tasks (*i.e.,* program comprehension and code writing) activate specific brain regions. For this reason, with our study we want to verify the correlation between cognitive human aspects and the quality of coding tasks in terms of correctness, time and readability. To do this, we measure cognitive human aspects through the use of psychometric tests as other studies present in literature [163, 57].

Oliveira *et al.* [163] performed an experiment with 109 developers in which they studied if developers can detect API security blindspots[2] in code. In addition, Oliveira *et al.* examined the influence of developer characteristics (*e.g.,* familiarity with code, cognitive function and personality) on the ability in detecting blindspots. Their results showed that there is no correlation between cognitive functions and the developer's ability to detect API blindspots. In a very recent study, Brun *et al.* [57] replicated the study by Oliveira *et al.* [163] and they obtained similar results. Differently from such studies, in this study we focus on the relationship between cognitive functions and the outcome of coding tasks, which are inherently different because they require developers to *writing* code, and not just *read* it.

In another study, Oliveira *et al.* [162] exploit the psychological manipulation to validate the following hypothesis: software vulnerabilities are not part of classical programming heuristics and developers do not consider them in their programming tasks. Results show that the security is not a principal activity of developers and this task need of a certain cognitive effort.

We can see that there are different studies on the cognitive human aspects in Software Engineering. The most interesting result is that some studies discover that during coding activities (*i.e.,* code writing and program comprehension) brain regions are activated during a program comprehension task related to working memory and attention level [173, 126, 207]. For this reason, in Chapter 8 we want to understand if there is a correlation between cognitive human aspects and the outcome of coding tasks in terms of time required to complete a task and the quality of the final solution (*correctness* and *code readability*).

[2] An API security blindspots is an error generated from the developer that can conduct to a violation of the API usage [63].

Part II

Code Readability Improvement & Evolution

*“Programs must be written for people to read,
and only incidentally for machines to execute.”*

Harold Abelson

CHAPTER 5

Why Do Developers Improve Source Code Quality?

Contents

5.1 Introduction

Software refactoring has been widely studied in the research community, with most of the works falling into three main research threads: (i) approaches aimed at identifying refactoring opportunities [167]; (ii) techniques to recommend refactoring solutions for a given design flaw [46]; and (iii) empirical studies looking at software refactoring from many different perspectives [231, 158, 208, 64, 229]. The knowledge of motivations pushing developers to perform refactoring [208] can help in building recommender systems able to propose suitable solutions for that. For this reason, understanding when and why developers perform refactoring has been the goal of many previous studies [231, 119, 44, 208, 229].

Some of these studies tried to answer this question by looking at specific factors that might correlate with refactoring operations, such as code quality proxies (*i.e.,* quantitative measures providing indications about the internal quality of code components, such as quality metrics or code smells) [44, 229]. While valuable, these studies provide limited insights into the reasons behind the performed refactorings, since their analysis is mostly quantitative and limited to a small number of factors. Other studies opted for a more qualitative approach by interviewing developers [231, 119] to identify the major factors that motivate their refactorings. Although these studies have pioneered the investigation of the reasons pushing developers to refactor their code, as observed by Silva *et al.* [208], the previously mentioned surveys are general purpose, meaning that they do not ask developers to *justify* specific refactorings they performed, but rather study refactoring habits in general. To address this limitation, Silva *et al.* [208] interviewed developers who authored 222 refactoring-related commits to understand the reasons behind these specific operations.

Stemming from the studies discussed above and to generalize their findings [231, 119, 208], this study describes a large-scale mining study combining quantitative and qualitative analyses to investigate the motivations behind refactoring operations, by observing code and discussions rather than interviewing developers. From a quantitative point of view, we mine the change history of 150 Java repositories hosted on GitHub to extract 287,813 refactoring operations of 25 different types performed by developers through the RMiner tool [222]. Then, we analyze product- (*e.g.,* slopes indicating whether the quality of code components as assessed by quality metrics is decreasing over time) and process-related (*e.g.,* source code change- and fault-proneness) factors that contribute to trigger refactoring actions. As compared to previous work [44, 229], we consider a more comprehensive set of factors and, more importantly, analyze them in a single model rather than in isolation, showing which ones are related to refactoring operations. From a qualitative point of view, we use the same set of systems to manually analyze a sample of 551 pull requests (PRs) in which (i) developers discuss refactoring **and** (ii) RMiner identifies at least one refactoring operation. Through a manual analysis, we identify the rationale of the refactoring change, and whether it is the main intent of the change or, rather, they are triggered by the code review process of the PR. As the main contribution of this analysis, we defined an extensive taxonomy of 67 motivations pushing developers to implement refactoring operations. Our qualitative analysis complements and generalizes the findings in previous survey-based studies [231, 119, 208] by investigating the same research question with a completely different experimental design.

As compared to the most similar work (*i.e.,* Silva *et al.* [208]), the following notable differences can be highlighted for what concerns the study design and findings:

- *Study Design: surveying developers vs analyzing their activities.* While Silva *et al.* [208] contacted the developers authoring the refactorings asking their motivations for the implemented changes, we manually inspect pull requests implementing refactorings by analyzing their discussion and related commits in order to create our taxonomy of motivations. Investigating the same research question with two different experimental designs can lead to additional insights, and helps in generalizing previous findings.

- *Study Design: complementing qualitative and quantitative analysis.* In our work, we analyze the motivations behind refactoring operations not only from a qualitative perspective (as done by Silva *et al.* [208]), but also by quantitatively studying the influence of product and process metrics on the triggering of refactoring operations. Also, to the best of our knowledge, our study is the first one analyzing these metrics in a single model rather than in isolation (as done in [44], for example).

- *Findings: complementing and generalizing Silva et al. [208].* As output of their study, Silva *et al.* [208] defined a list of 44 motivations for 12 frequently applied refactoring operations. Our taxonomy, besides confirming 41 of their motivations, thus improving the generalizability of their findings, includes 26 additional ones that are not covered in the previous study.

Our quantitative analysis indicates that code readability and process-related factors correlate with the changes a commit containing refactoring operations has. As the main result of the qualitative analysis, we provide a comprehensive taxonomy of 67 categories of motivations leading developers to refactoring operations. We describe and exemplify each category, and discuss its implications in refactoring research and practice.

Motivated by such results, as it will presented in the next Chapters, we want to study code readability in the context of software evolution. Thus, we want to understand (i) the importance of readability for developers (Chapter 6); (ii) the evolution of code readability in complex software systems (Chapter 7); (iii) the influence of developers' personal characteristics on the evolution of code readability (Chapter 8).

This chapter is organized as follows. Section 5.2 describes the study design we conducted to perform the quantitative and qualitative analysis. Section 5.3 describes obtained results from our study. Finally, in Section 5.4 we report threats to validity of our study and in Section 5.5 we conclude this chapter.

5.2 Design of the Study

The *goal* of this study is to quantitatively and qualitatively analyze the context in which refactoring operations occur in open source projects, with the aim of identifying the circumstances that may make a refactoring happen. The *quality focus* relates not only to code quality, but also to the improvement of the software development process. The *context* consists of 287,813 refactoring actions automatically identified in 150 open-source projects and, for the qualitative analysis, of 551 manually-analyzed PRs mentioning refactoring operations and linked onto refactoring-related commits.

We address the following two research questions (RQs):

RQ$_1$: *Which product and process-related factors relate with an increase of refactoring operation chances?* We are interested in studying if various source code features or process features correlate with the presence of refactoring operations in a commit.

RQ$_2$: *What are the reasons for performing a refactoring operation?* We investigate the rationale behind refactoring opportunities. We consider refactorings occurred in PRs, and perform a qualitative analysis of developers' discussions over the PR. Also, since previous work found that most refactoring operations occur with other changes [158], by analyzing PRs we give a closer look at this phenomenon, investigating if the refactoring was tangled with other changes, and looking at whether the refactoring was the primary purpose of the PR. We decided to answer this RQ by looking at PRs rather than at commits implementing refactorings since PRs offer a richer set of information to analyze to derive the rationale behind refactoring operations. Indeed, they often feature a discussion among developers that can help in better understanding what the goal of the implemented change was.

The formulated RQs investigate the same phenomenon (*i.e.,* what the motivations for refactoring operations are) from two different perspectives (quantitative—RQ$_1$ *vs* qualitative—RQ$_2$). The catalog of motivations identified in the two RQs can complement and support each other.

5.2.1 Study Context

We identified the projects to be studied among repositories hosted on GitHub. Since the infrastructure used in our study (*e.g.,* the refactoring detection tool) only supports Java, we focus on Java projects. Among all Java projects on GitHub, we aim at studying active projects having a non-trivial change history to study (to mine the PRs needed for our study) and not representing personal and/or toy projects (*e.g.,* a project created by a student during an assignment). To identify

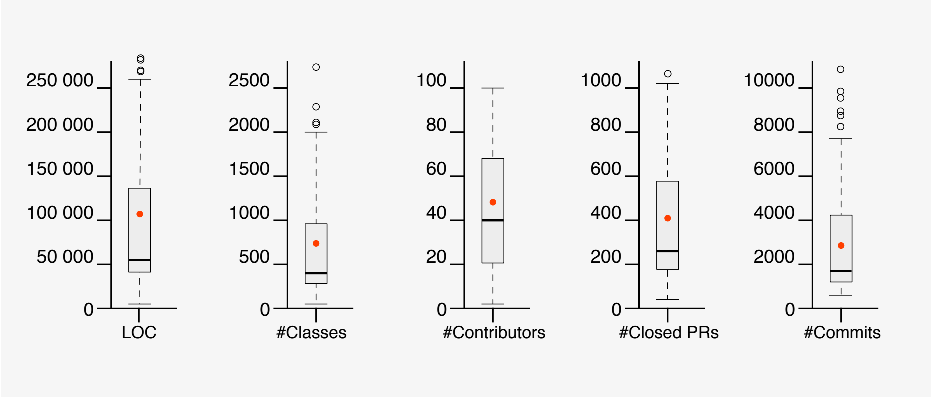


Figure 5.1: Characteristics of the 150 projects used in our study

these projects we applied a number of selection criteria, only retaining projects having:

- *At least 5 contributors and 1 fork*, to exclude personal/toy projects.

- *At least 500 commits and 100 PRs*, to exclude projects having a short change history and unlikely to provide useful PRs for our study.

- *Modified at least once in the period Jan-May 2019*, to exclude inactive projects at the time in which this study has been run.

From the set of 303 remaining projects, we randomly selected 150 of them for our study (list available in [34]). The choice of selecting a subset of the 303 projects was dictated by the computationally expensive data extraction process adopted in our study. Indeed, as detailed in the following, besides detecting refactoring operations, we computed 42 product- and process-metrics (*e.g.,* code quality metrics, change-proneness of classes) for each of the 213,102 commits in the studied projects. This process took three months on a 56-core server. Figure 5.1 reports boxplots depicting the distribution of Lines of Code (LOC), number of classes (#Classes), number of contributors (#Contributors), number of closed PRs (#Closed PRs), and number of commits (#Commits) for the analyzed 150 systems. The raw data from which this figure has been created is available in our replication package [34].

We used the RMiner tool [222] to detect the refactorings implemented by developers in the studied projects. We focus on commits performed in the master/default branch of each project. We have chosen RMiner due to its high reported precision (98%) and recall (87%) [222]. RMiner takes as input two consecutive commits and provides as output the set of detected refactorings (see [222] for the supported refactorings).

5.2.2 Quantitative Analysis (RQ_1)

The occurrences of the detected refactorings constitute the *dependent variable* for RQ_1. As *independent variables*, we consider process-/product-related factors for each snapshot s_i (commit) of the master branch.

Identification of Product and Process Metrics

The considered metrics are summarized in Tables 5.1, 5.2 and 5.3 and described in the following. The selection of these metrics (detailed in the following) is based on the will to include in our study:

1. *Metrics capturing code quality from different perspectives (Table 5.1).* We included both structural and semantic (*i.e.,* textual) metrics that have been shown to capture orthogonal code quality aspects [144]. Also, we considered the recent readability metrics proposed in the literature [59, 199], that have been shown to highly correlate with the developers' assessment of code readability.

2. *Code smells and quality issues widely studied in the literature (Table 5.2).* The presence of code smells has been correlated with higher change- and fault-proneness of code [166] and, thus, they could also be responsible for the triggering of refactoring actions. Also, static analysis tools are more and more used in the context of continuous integration to perform basic code quality checks at commit time. Thus, we decided to include the warnings raised by one of these state-of-the-art tools, *i.e.,* PMD [17].

Table 5.1: Quality Metrics (Product-related factors)

Near each factor, we indicate whether (✓) it was retained.
The factors retained in the model are also highlighted in boldface.

Metric	**Description**
CBO ✓	Coupling Between Object classes: measures the dependencies a class has [70]
WMC ✓	Weighted Methods per Class: sums the cyclomatic complexity of the methods in a class [70]
RFC ✓	Response For a Class: the number of methods in a class plus the number of remote methods that are called recursively through the entire call tree [70]
ELOC ✓	Effective Lines Of Code: the lines of code excluding blank lines and comments
NOM ✓	Number Of Methods in a class
NOPM ✓	Number Of Public Methods in a class
DIT ✓	Depth of Inheritance Tree: the length of the path from a class to its farthest ancestor [70]
NOC ✓	Number Of Children (direct subclasses) of a class
NOF ✓	Number Of Fields declared in a class
NOSF ✓	Number Of Static Fields declared in a class
NOPF	Number Of Public Fields declared in a class
NOSM ✓	Number Of Static Methods in a class
NOSI ✓	Number Of Static Invocations of a class
HsLCOM ✓	Henderson-Sellers revised Lack of Cohesion Of Methods (LCOM): a class cohesion metric based on the sharing of local instance variables by the methods of the class [70]. HsLCOM dresses limitations of the original LCOM [110]
C3 ✓	Conceptual Cohesion of Classes: avg. textual similarity between all pairs of methods in a class [144]
StrRead ✓	Structural readability: uses structural aspects (*e.g.,* line length) to model code readability [59]
ComRead ✓	Comprehensive readability model: combines structural, visual (*e.g.,* alignment) and textual features (*e.g.,* comments readability) [199]

3. *Process-related factors (Table 5.3).* These metrics are meant to provide a view on the development process, the developers involved in it, and historical information about the code components. We conjecture that these factors can play an important role in taking refactoring decisions, as also partially confirmed by previous work in the literature [229].

As detailed in Section 5.2.2, to avoid multicollinearity, we performed a variable selection. Near each metric, we indicate the cluster it belongs to and whether (✓) it was retained.

Source Code Quality Metrics. We consider, for each class C changed in each snapshot s_i, its quality *trend* as assessed by the 18 metrics in Table 5.1. These metrics capture different aspects of code quality, including size (*e.g.,* ELOC), coupling (*e.g.,* CBO), inheritance (*e.g.,* DIT), complexity (WMC), encapsulation (*e.g.,* NOPM), and readability (*e.g.,* StrRead). The first 13 metrics in Table 5.1 (*i.e.,* until NOSI included) have been computed by using the CK tool [4]. For the HsLCOM and C3, we used our implementation, while for the readability metrics we relied on the original implementations of the tools computing these metrics kindly made available by the original authors of the papers that introduced them [59, 199]. We start by measuring these 18 metrics on each class in each mined snapshot. Then, based on this information, we compute, for each snapshot, the slope of each metric over a window of N preceding commits (we set N=10 according to a previous work recommending just-in-time refactoring [169]). The slope of a line describes its steepness and in our case can highlight, for example, continuing degradation of some quality aspects (*e.g.,* a high positive slope for the WMC metrics indicates a steep increase in complexity for a class over time). Thus, using slopes we capture the improvement or degradation of quality factors, where the latter may trigger a refactoring. Clearly, slopes were considered unavailable for the first ten commits of a class.

Code Design Flaws and Quality Warnings. We consider code design flaws related to the lack of adoption of good Object-Oriented coding practices (*i.e.,* Spaghetti Code, Excessive Coupling), to complex/large code components (*i.e.,* Blob Class, Complex Class) as well as other design flaws and warnings (*i.e.,* Excessive Imports, Too Many Methods) raised by a static analysis tool. We detect five types of code smells using an implementation of the DECOR smell

Table 5.2: Code Design Flaws (Product-related factors)

Near each factor, we indicate whether (✓) it was retained.
The factors retained in the model are also highlighted in boldface.

Design Flaw	**Description**
DECOR Code Smells	
Blob Class	A large class that monopolizes most of the application logic [56]
Complex Class ✓	A class characterized by a high cyclomatic complexity [56]
Spaghetti Code ✓	A class declaring long methods without parameters [56]
CDSBP ✓	Class Data Should be Private: violation of information hiding principle [97]
Functional Decomposition ✓	Scarcely used object-oriented principles, such as inheritance and polymorphism; few methods and many private fields [56]
PMD code quality warnings	
Excessive Coupling ✓	A highly coupled class hindering reuse and maintainability [97]
Too Many Nested If Statements ✓	Makes the code harder to understand and increase error-proneness
Excessive Imports ✓	It might indicate too high coupling
Too High NPath Complexity	NPath is the number of acyclic execution paths throughout a method
Excessive Method Length ✓	It might indicate too many functionalities in a single method
Excessive Class Length	It might indicate too many responsibilities implemented in a class
Too Many Fields ✓	It can make the code hard to understand
Too Many Methods ✓	It might indicate too many responsibilities in a class
Cyclomatic Complexity	An excessive degree of decisional logic in a class
Excessive Parameter List ✓	It might indicate the need for a new object to wrap them
NCSS Type Count ✓	Similar to excessive class length, but it only considers actual statements
NCSS Method Count ✓	Similar to excessive method length, but it only considers actual statements
NCSS Constructor Count ✓	Equivalent of NCSS Method Count for constructors

detector based on the original rules defined by Moha *et al.* [151]. The choice of using DECOR is driven by the fact that (i) it is a state-of-the-art smell detector having high accuracy in detecting smells [151]; and (ii) it applies simple detection rules that allow it to be very efficient. The latter was a strict requirement for our analysis since we detected smells in all classes and for all studied systems' snapshots. In addition, we also consider 13 flaws from a widely-used static analysis tool that does not require code compilation, *i.e.,* PMD [17]. The set of detected design flaws and code quality warnings is described in Table 5.2.

Process-related factors. Besides the product-related factors previously described, we also study how process-related factors correlate with refactoring. In this case, we extract for each analyzed snapshot the factors summarized in Table 5.3.

Given a snapshot s_i, we compute its distance (in commits) from the previous and next release (first two rows in Table 5.3). This to verify the conjecture of Vassallo *et al.* [229] that refactoring does not occur immediately before/after a release. This information was retrieved using the GitHub API, through which it is possible to access all the tags related to a project. Then, we manually looked at the tags assigned to each project to isolate the ones referring to a new release.

We also consider the change- and fault-proneness of classes. The change-proneness is computed as the ratio between the total number of lines changed in the class C from the date of its addition to the project and the total number of commits in which C was changed, until each snapshot s_i.

The fault-proneness for C is computed as the number of bug-fixing commits it has been subject to in the past (*i.e.,* before s_i). For each project, we firstly identified all bug-fixing commits by matching patterns [91]: "fix" or "solve" or "close" **and** "bug" or "defect" or "crash" or "fail" or "error". Then, for a given class C and for each snapshot s_i, we compute the number of bug-fixing commits preceding s_i and impacting C. Section 5.4 discusses the extent to which this simple heuristic for identifying bug fixes leads towards imprecisions.

Finally, we consider two metrics capturing the experience of the developers who worked on the system's classes. The first metric, named *Developer Overall Experience*, assesses the experience of each developer as the number of commits she performed in the past. For each snapshot s_i and for each of its classes C,

Table 5.3: Process-related factors

Near each factor, we indicate whether (✓) it was retained.
The factors retained in the model are also highlighted in boldface.

Metric	Description
Closeness to a previous release ✓	The number of commits until the previous minor/major release
Closeness to a next release ✓	The number of commits until the next minor/major release
Fault-Proneness ✓	Number of bugs fixed in the project history on a given class
Change-Proneness ✓	The average number of lines impacted in commits related to a class
Developer Overall Experience ✓	The number of past commits a developer performed
Developer Class Experience ✓	The number of past commits on a class performed by a developer

we extract the list of developers who modified C in the past (*i.e.,* before s_i). For each commit c_j (with $j < i$) in which C has been modified, we compute the experience of the developer authoring c_j (*i.e.,* the number of commits she performed *before* c_j). This gives us a distribution of developers' experiences, for which we compute the minimum. Indeed, the minimum represents the lowest experience of a developer who worked on C, and we assume it might be correlated with future refactoring actions taken on C.

The *Developer Class Experience* computes a class-related experience: for each snapshot s_i and for each of its classes C, this form of experience is computed for a given developer as the number of commits impacting C she performed in the past. Thus, it is a more specific version of the overall experience. We compute this metric for each s_i and C under study in the same way explained for the overall experience.

Metrics Aggregation and Preprocessing

Since in RQ_1 we are interested to build an *explanatory model* explaining which factors correlate with the presence of refactoring actions in a snapshot, we had to aggregate metrics for all classes involved in each snapshot. For the product metrics, we compute the maximum slope among all classes involved in the snapshot, except for the conceptual cohesion (*C3*) and readability (*StrRead* and *ComRead*) metrics, which go in the opposite directions than other metrics (higher values are better). In such cases, we consider the minimum. In both cases, the rationale is to identify the “worst case" in a snapshot, which could ideally trigger a refactoring. As for the DECOR smells, we count the number of classes exhibiting a smell in each snapshot, while for PMD we sum the number of warnings of each type among changed classes. Similarly to what done for product metrics, for process metrics, we compute the maximum (*e.g.,* maximum number of bugs), except for the experience-related metrics, where we consider the minimum, again to consider the worst-case scenario. Finally, release-related metrics do not need to be aggregated, since they are already at commit granularity.

After that, to avoid multi-collinearity, we use the *R redun* function of the *Hmisc* package [108] for removing redundant variables. The *redun* function stepwise removes variable, starting from the most predicable one, until no variable can be predicted with an adjusted R^2 greater than a given threshold (0.8 in our study). Once again, we use the whole dataset to perform correlation analysis, because we intend to build an explanatory model and not a predictive model.

Since the value of our independent variables can depend on projects’ characteristics, and to properly interpret the importance of each variable in the model, we normalize variable values, within each project, in the interval [0, 1]. This is done by subtracting the minimum and dividing by the difference between the maximum and minimum. Finally, to build a model easy to be interpreted, we invert (*i.e.,* compute 1-x) the values of variables going towards a different direction than the others (*i.e.,* those for which the higher the better).

Mixed-model Building

Once variables have been preprocessed, we address $\mathbf{RQ}_1$ by building mixed-effect generalized linear models. The model, built using the *glmer* function of the *lme4* [41] R package, is a logistic regression mixed-effect model where: (i) the dependent variable is a dichotomous variable indicating whether at least a refactoring was performed in a given commit; (ii) the independent variables (fixed effects) are all the aforementioned ones, after having pruned out those highly correlating with others; (iii) the random effect is the project in which the change occurred. The latter aims at controlling within-project effects, *e.g.,* a project following a specific development process had better code quality assurance policies than others. To simplify, our model reports whether the status of the system (as assessed by the used independent variables) in the snapshot S_{i-1} triggered a refactoring in the subsequent commit C_i.

To answer $\mathbf{RQ}_1$, we report the details of the model, among others the coefficient of each factor in the model, and the p-value indicating whether the factor is statistically significant or not (for a significance level of 95%). We also report the odds ratio (OR) which, for a logistic regression model, is given by e^{c_i} where c_i is the coefficient of the i-th factor. An OR > 1 indicates that a unity increase of a variable increases OR times the chances of a refactoring to occur.

5.2.3 Qualitative Analysis of Refactoring Discussions in Pull Requests (RQ$_2$)

For the qualitative analysis, we identified PRs likely discussing refactorings using two criteria to be satisfied: (i) whether a commit is part of a PR or made during its review contains a refactoring identified by RMiner, and (ii) whether the PR title or comments contain refactoring-related keywords. We used a list of refactoring keywords defined in a previous work [61] (available in our replication package [34]) and augmented it with all names of refactorings identified by RMiner [222]. Note that, while this selection process can generate false positives (*i.e.,* PRs unrelated to refactoring operations), these will be discarded during the manual analysis and, thus, do not represent a source of noise for our study.

Once the candidate set of 2,400 PRs has been identified, we created a randomly-stratified sample of 551 PRs. The strata here were represented by the projects, *i.e.,* PRs were sampled across projects proportionally based on the number of candidate PRs found in the previous step. The total number of PRs sampled allows us to ensure a significance interval (margin of error) of ±5% with a confidence level of 99%, and feature a total of 8,108 refactoring operations identified by RMiner. This estimation has been performed using a sample size (SS) calculation formula for an unknown population [193]:

$$SS = p \cdot (1 - p)\,\frac{Z_{\alpha}^{2}}{E^{2}}$$

and SS_{adj} for a known population *pop*:

$$SS_{adj} = \frac{SS}{1 + \frac{SS-1}{pop}}$$

where p is the estimated probability of the observation event to occur (we assume it being 0.5 if we don't know it a priori), Z_{α} is the value of the Z distribution for a given confidence level, and E is the estimated margin of error (5%).

We then uploaded the sample of PRs on a tagging webapp we used to perform a manual coding of PRs. The webapp presented to the annotator the following information: (i) the PR title and hyperlink to the discussion; (ii) the refactoring-related keyword(s) matched in the PR text; and (iii) the list of refactorings detected by RMiner in commits linked to the PR, as well as the links to the GitHub diff pages of the commits themselves.

Through the coding app, each annotator could add one or more tagging items, containing the following information: (i) the type of refactoring action performed and discussed in the PR, or whether the change discussed was related to a combination of refactorings; (ii) whether the refactoring was the original intent of the PR, whether it happened as a consequence of the PR discussion, or whether it happened accidentally because of another change; (iii) whether the refactoring was tangled with other changes, or if it was the only purpose of the PR; (iv) finally, a tag indicating the motivation behind the refactoring, as it could be inferred from the inspection of the PR title/description, from its discussion, and from the commits related to it, looking at commit messages and, when needed, code diff. Note that each annotator could add more than one motivation for each PR (*e.g.,* one for each refactoring operation, or even more than one for the same refactoring). To assign the tag describing the motivation, the annotator could choose an available tag in a drop-down menu (from those previously created by other annotators or by herself), or add a new one if no tag was fitting the specific case. If an annotator realized that the PR discussion was not related to refactoring, the PR was tagged as "false positive".

Six of the seven authors of the study took part in the annotation process. The webapp we developed took care of automatically assigning each PRs to at least two of the involved annotators. We collected a total of 1,223 tags each one reporting a motivation for a refactoring (or combination of refactorings) performed in a PR. After each PR was tagged by two annotators, three of the authors jointly worked on the available tags to perform a card sorting activity [213] aimed at merging duplicates (*i.e.,* similar tags having the same meaning), and started grouping tags into categories. Then, we created a first taxonomy describing the different purposes of refactorings by only using the 699 tags for which there was no conflict (*i.e.,* the same tag was used by the two annotators for motivating the refactoring observed in a PR). After a first draft of the taxonomy was produced, two different authors refined it, by renaming some categories and moving sub-categories through the taxonomy. Once the final taxonomy was produced, three authors jointly discussed the conflicting cases in the categorization (524 out of 1,223 tags) and assigned them to suitable taxonomy categories, creating new ones when needed, and ensuring a consistency of category naming.

To address $\mathbf{RQ}_2$, we report and discuss the taxonomy of refactoring motivations inferred as previously explained. In particular, we discuss the various categories, highlighting the percentages of PRs belonging to the category, reporting some examples, and highlighting the implications resulting from our empirical findings.

5.3 Results

In the following we report and discuss the results addressing our RQs (Section 5.2).

5.3.1 Which Product and Process-related Factors Relate with an Increase of Refactoring Operation Chances?

Over the 213,102 snapshots analyzed, RMiner identified a total of 287,813 refactoring operations. More in details, our dataset contains 35,560 commits

Table 5.4: Generalized mixed effect logistic regression model: diagnostics, residuals, and random effect

Diagnostics					
	AIC	**BIC**	**logLik**	**deviance**	**df resid.**
	21,071.5	21,362.4	-10,499.7	20,999.5	23,860
Scaled residuals					
	Min	**1Q**	**Median**	**3Q**	**Max**
	-2.0565	-0.5256	-0.3867	-0.1094	11.3848
Random effects					
Groups Name	**Variance**	**Std.Dev.**			
ProjectName (Intercept)	1.549	1.245			

($\simeq$ 17%) with at least one refactoring operation. If we exclude renaming operations (*Rename Method* and *Rename Class*), RMiner found a total of 209,385 refactorings in 28,716 different snapshots (14%).

Tables 5.4 and 5.5 report the results of the logistic regression mixed-effect model. More specifically, Table 5.4 reports the model diagnostics (Akaike Information Criterion — AIC [27], Bayesian Information Criterion — BIC, log likelihood, deviance, and degree of freedom residuals), the scaled residuals, and the random effect (project estimate). Concerning the model fitting (Table 5.4), we tried different models, namely logistic (*i.e.,* the one reported, AIC=21,071), linear (AIC=22,565), and Poisson (AIC=22,560). Also, although the analysis performed using the *redun* function already used a goodness-of-fit to iteratively remove variables, we experimented logistic models using structural metrics only (AIC=89,751), conceptual metrics only (AIC=89,475), code design flaws only (AIC=179,528), and process metrics only (AIC=23,583). Ultimately, the comprehensive logistic regression model we report is the one with the smallest AIC among those considered.

Table 5.5 reports the OR, estimate, standard error, z-value and p-value for the various factors we considered. We report in bold face the coefficient for which there is a statistically significant correlation. Metrics that have been inverted (*e.g.,* C3) are named with the prefix “Lack”.

Table 5.5: Generalized mixed effect logistic regression model: effect of considered factors

	Metric	**OR**	**Estimate**	**Std. error**	***z*-value**	***p*-value**
	(Intercept)	0.00	-7.52	0.52	-14.50	<**0.01**
Quality	**LackStructRead**	**3.14**	1.14	0.38	3.05	<**0.01**
	LackComRead	**2.68**	0.99	0.41	2.42	**0.02**
	LackC3	**1.87**	0.63	0.30	2.06	**0.04**
	CBO	1.02	0.02	0.03	0.66	0.51
	WMC	0.98	-0.02	0.01	-1.57	0.12
	DIT	1.17	0.16	0.12	1.39	0.17
	NOC	0.95	-0.06	0.28	-0.20	0.84
	RFC	0.98	-0.02	0.02	-1.05	0.29
	NOM	**0.88**	-0.12	0.05	-2.54	**0.01**
	NOPM	**1.22**	0.20	0.06	3.19	<**0.01**
	NOSM	0.91	-0.09	0.12	-0.76	0.45
	NOF	1.12	0.11	0.08	1.34	0.18
	NOSF	0.89	-0.11	0.13	-0.88	0.38
	NOSI	1.01	0.01	0.07	0.12	0.90
	LOC	1.00	0.00	0.00	0.91	0.36
	HsLCOM	**1.94**	0.66	0.29	2.28	**0.02**
Code Design Flaws	IsGodDecor	1.12	0.12	0.14	0.81	0.42
	IsCDSBPDecor	0.90	-0.11	0.15	-0.71	0.48
	IsComplexDecor	1.03	0.03	0.14	0.23	0.82
	IsFuncDecDecor	0.76	-0.28	0.25	-1.11	0.27
	IsSpaghCodeDecor	1.10	0.09	0.13	0.69	0.49
	AvoidDeeplyNestedIfStmts	0.92	-0.09	0.20	-0.43	0.67
	CouplingBtwObjects	0.73	-0.31	0.22	-1.42	0.16
	ExcessiveImport	1.04	0.04	0.20	0.18	0.86
	ExcessiveMethodLength	0.96	-0.04	0.23	-0.18	0.85
	ExcessiveParameterList	1.28	0.25	0.20	1.22	0.22
	TooManyFields	1.08	0.08	0.20	0.40	0.69
	TooManyMethods	0.66	-0.42	0.29	-1.42	0.16
Process	**LackGeneralExp**	**1.26**	0.23	0.10	2.30	**0.02**
	LackFileExp	**8.93**	2.19	0.13	16.45	<**0.01**
	FilesRelatedToIssueFix	**2.09**	0.74	0.07	10.12	<**0.01**
	AvgLinesImpactedInCommit	**1.93**	0.66	0.17	3.96	<**0.01**
	DistancePreviousRelease	1.13	0.12	0.08	1.59	0.11
	DistanceNextRelease	**1.43**	0.36	0.09	4.12	<**0.01**

Looking at code quality metrics, we found that the lack of structural readability plays a significant role: lack of structural readability [59] increases the odds of refactoring (OR=3.14). At the same time, *ComRead* readability metric and *LackC3* show a marginal significance (p-value = 0.02 and p-value = 0.04), respectively. In particular, looking at the *ComRead* readability metric combining structural and textual features [199] the OR is 2.68, while classes showing a decrease in their conceptual cohesion (*LackC3*) have 1.87 times higher odds of being refactored.

Among the structural metrics, we found that *NOM*, *NOPM* and *HsLCOM* have a statistically significant effect (marginally significant for *HsLCOM*), although the

OR for *NOM* and *NOPM* is close to one. Instead, the OR for *HsLCOM* is 1.92, indicating that, as expected, a lack of cohesion increases the odds of inducing a refactoring operation.

In conclusion, from our analysis it results that conceptual and readability metrics play a more important role in the model than structural metrics. This finding is aligned with previous work aimed at applying conceptual metrics to suggest software refactoring [42] and modularization [47], and with findings of the seminal work about C3, indicating that such a metric is complementary to structural metrics [145].

None of the design flaws plays a statistically significant role. Although we expect that developers take care of removing smells, or try to "make static analysis tools happy", and although previous work has pointed the role of refactoring for improving code having a poor quality, *e.g.,* overly complex code [231, 122, 208, 44], an evolutionary study on code smells indicates that smells mostly disappear when the source code is being rewritten, and only in less than 10% of the cases because of a refactoring action [224].

Interestingly, process-related metrics are highly representative if compared to product-related metrics: five of the six considered process-related metrics are statistically significant. Previous bug fixes play a role: a unit increase of the *FileRelatedToIssueFix* factor results in 2.09 higher odds of applying a refactoring in the system. Not only classes subject to bug fixes are likely to be fault-prone in future [122, 153] but, since they are subject to (often quick-and-dirty) patches, they may necessitate refactoring actions. For related reasons, classes changing a lot (*AvgLinesImpactedInCommit*) also need to be refactored, although the OR is smaller (1.93).

Moving the attention to the metrics capturing the developers' experience, the *Developer Class Experience* (*LackFileExp*) has the highest OR. A unit increase of this factor related to the lack of specific experience (in terms of past commits) of developers that have recently modified a class, and therefore a decrease of experience increases the odds of refactoring by 8.93 times. In other words, changes applied by developers with little knowledge about a code component increase the need for restructuring it in the future. The general experience also plays a statistically significant role, although the OR is relatively small (1.26).

Finally, looking at the proximity to a release (*DistanceNextRelease* and *DistancePreviousRelease*), the metrics indicate that refactoring operations are applied to the system far from a release of the system. More specifically, increasing the number of commits to a subsequent release, there are 1.43 higher odds of applying a refactoring. However, results are not significant while looking at the number of commits from a previous release (*i.e.,* p-value = 0.11). Our findings confirm previous literature [120, 229] since developers are aware that some kind of refactorings may result in the introduction of new faults [43] and in any case, refactoring represents a costly and risky operation [120]. For this reason, it is not very common to apply refactoring close to a new release of the software product. Furthermore, once released a new version of the software, developers likely tend to focus on bug-fixing activities instead of applying refactoring operations. In summary, based on our observations, refactorings are less likely to occur either immediately before major releases (developers focus on new features and, for what possible, on reliability of what they release), and immediately after (developers work on bug fixes). Instead, refactorings are more likely to happen in-between. Once again, note that this conclusion is based on purely observational data, and distance from the next release is unlikely to be used for prediction purposes (developers do not know when the next release is, unless a project or organization adopts very rigid time-fixed releases).

We conclude RQ$_1$ stating that, on the one hand, by observing product metrics, only code readability plays a significant role. On the other hand, process-related metrics play a significant role. These are metrics related to previous changes and bug fixes, and to the experience of recent change authors.

5.3.2 What are the Reasons for Performing a Refactoring Operation?

Before digging into the results, Table 5.6 reports statistics about the refactoring operations we found in the analyzed PRs. Note that: (i) several refactorings can be applied in one PR, therefore the number of refactorings is higher than that of PRs; (ii) we only list refactoring operations we observed at least 10 times. However, the overall number of refactorings (1,117) also includes the

Table 5.6: Statistics of refactoring operations labeled in the 551 analyzed PRs.

The ‘Freq.” column reports the number of times that the annotators defined a tag explaining the rationale behind each specific type of refactoring operation. The total number of 1,117 tags is the result of the 1,223 tags we defined, excluding the 94 “unclear” (*i.e.,* cases in which the annotators did not manage to identify the rationale for the refactoring) and 12 “false positives” (*i.e.,* PRs that were unrelated to refactoring).

Refactoring operation	**Freq.**	**When?**			**Tangled**	
		Orig. intent	**Collateral**	**After discuss.**	**Yes**	**No**
Combination of refact.	578	65%	3%	32%	86%	14%
Extract operation	112	62%	3%	35%	62%	38%
Rename method	69	47%	4%	49%	59%	41%
Rename class	53	39%	6%	55%	56%	44%
Move class	39	64%	8%	28%	49%	51%
Extract variable	29	52%	35%	14%	72%	28%
Rename variable	29	35%	14%	52%	83%	17%
Extract interface	27	59%	11%	30%	59%	41%
Rename attribute	22	35%	8%	57%	44%	56%
Extract and move	22	68%	0%	32%	55%	45%
Rename parameter	19	69%	5%	26%	37%	63%
Move attribute	15	53%	14%	33%	73%	27%
Move operation	13	69%	0%	31%	39%	61%
Extract superclass	10	70%	10%	20%	80%	20%
Overall	**1117**	**60%**	**5%**	**35%**	**73%**	**27%**

instances related to refactoring types we do not show in Table 5.6 (since having less than 10 occurrences). In most cases, developers do not discuss a specific refactoring operation. Instead, they rather provide a rationale for a combination of refactorings (∼52% of the cases). Then, *extract operation* (∼10%) and renaming refactorings in general (∼17%, in total) are the ones more discussed by developers. Surprisingly, we found that only a small percentage (5%) of refactorings were done collaterally, *i.e.,* without mentioning them at all. Instead, many of them were done as the original intent of the PR (∼60%) or after discussing with other developers (∼35%).

Some refactoring operations, such as *extract variable* and *rename variable*, were performed collaterally more often, given their "local" nature: a variable name only matters in the methods in which it is declared, while a class name can possibly impact the whole system. It is worth noting that developers perform most of the renaming operations after they receive feedback from their peers. This shows that names are often discussed in PR reviewing activities. Finally, in line with Murphy-Hill *et al.* [158], we found that about a fourth of the refactorings are tangled with other changes.

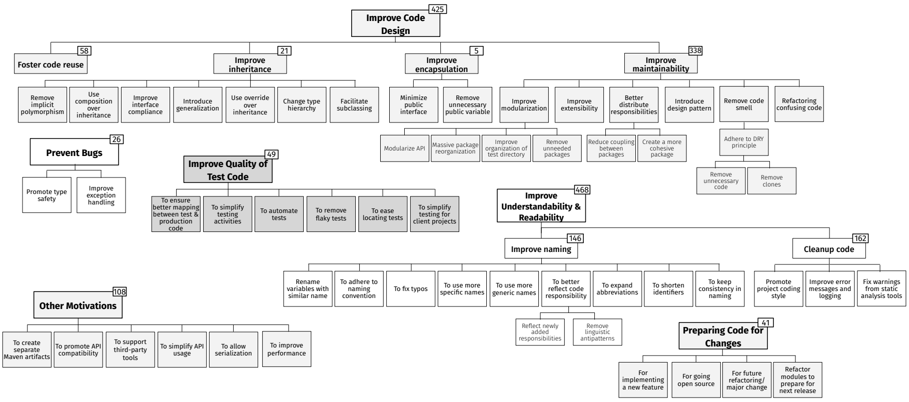

Improve Code Design
425
Foster code reuse
58
Remove implicit polymorphism
Use composition over inheritance
Improve inheritance
21
Improve interface compliance
Introduce generalization
Use override over inheritance
Change type hierarchy
Facilitate subclassing
Improve encapsulation
5
Minimize public interface
Remove unnecessary public variable
Improve maintainability
338
Improve modularization
Modularize API
Massive package reorganization
Improve organization of test directory
Remove unneeded packages
Improve extensibility
Better distribute responsibilities
Reduce coupling between packages
Create a more cohesive package
Introduce design pattern
Remove code smell
Adhere to DRY principle
Remove unnecessary code
Remove clones
Refactoring confusing code
Prevent Bugs
26
Promote type safety
Improve exception handling
Improve Quality of Test Code
49
To ensure better mapping between test & production code
To simplify testing activities
To automate tests
To remove flaky tests
To ease locating tests
To simplify testing for client projects
Improve Understandability & Readability
468
Improve naming
146
Rename variables with similar name
To adhere to naming convention
To fix typos
To use more specific names
To use more generic names
To better reflect code responsibility
Reflect newly added responsibilities
Remove linguistic antipatterns
To expand abbreviations
To shorten identifiers
To keep consistency in naming
Cleanup code
162
Promote project coding style
Improve error messages and logging
Fix warnings from static analysis tools
Other Motivations
108
To create separate Maven artifacts
To promote API compatibility
To support third-party tools
To simplify API usage
To allow serialization
To improve performance
Preparing Code for Changes
41
For implementing a new feature
For going open source
For future refactoring/ major change
Refactor modules to prepare for next release

Figure 5.2: Motivations behind refactoring operations

Figure 5.2 depicts the taxonomy of refactoring motivations we have identified. It comprises six root categories: (i) *Improve Code Design* groups refactoring operations targeting an improvement of the system design, *e.g.,* by fostering the reusability of code; (ii) *Improve Understandability & Readability* includes refactorings aimed at reducing the effort to read and understand code, *e.g.,* by renaming identifiers; (iii) *Improve Quality of Test Code* groups all refactorings performed to improve the quality of the test code or ease the testing process; (iv) *Prevent Bugs* identifies refactorings performed to prevent the future introduction of bugs; (v) *Preparing Code for Changes* includes refactorings performed in preparation of other changes, *e.g.,* refactoring the code before implementing a new feature; (vi) finally, *Other Motivations* groups those motivations that cannot be classified into one of the previous categories.

It is important to point out that some of the categories of the taxonomy are not mutually exclusive. For example, a refactoring aimed at improving code readability is also likely to improve maintainability. However, readability can be improved for multiple purposes (*e.g.,* simplify testing), and, for this reason, we separated these categories. We acknowledge that other choices in terms of categories and assignment of instances to these categories are possible. Also, the hierarchical organization of the categories only indicates that child categories are specialization of their parent categories, while it does not imply that two categories at the same hierarchy level represent motivations at the same level of abstraction (*e.g., Improve Inheritance* and *Foster Code Reuse* appear at the same level, but the former is a more concrete motivation as compared to the latter).

Figure 5.2 also reports for each category of "motivations" the number of PRs in which we found related refactoring operations. For readability purposes, we only report these numbers for the main categories. Note that the number for a parent category does not correspond to the sum of the children, because some PRs were only assigned to the parent category, as the motivation was not specific enough. Also, the sum of refactoring instances in all root nodes does not correspond to the total number of 551 manually analyzed PRs because some PRs comprise refactorings falling into multiple categories, and we labeled some refactorings as *Unclear* (94) and discarded 12 PRs as *False Positive*.

We compared our taxonomy with the list of 44 motivations derived by Silva *et al.* [208] for 12 frequently applied refactoring operations (see Tables 3 and 4 in [208]). In particular, two of the authors of the study tried to map Silva *et al.*'s motivations into our taxonomy, to see whether they were covered or not. Note that the mapping is not one-to-one since one motivation identified by Silva *et al.* [208] may be mapped to more than one category in our taxonomy, as well as one of our categories can group more than one of their motivations. This is expected since their motivations and the categories in our taxonomy have been derived by using two different methodologies. Indeed, while we have categorized the possible motivations behind the application of refactoring operations by looking at the discussions in PRs, Silva *et al.* [208] have asked the reasons behind specific instances of refactoring operations to the original developer who has applied it.

Only 3 out of the 44 motivations from Silva *et al.* [208] cannot be mapped in our taxonomy. The main reason is that for these three instances (*i.e., Enable recursion*, *Convert to top-level container*, and *Convert to inner class*) it was unclear to us the actual motivation behind the refactoring. For example, enabling recursion could be done to improve performance as well as to improve code readability. However, this high overlap between the two sets of motivations (i) validates and generalizes the work done by Silva *et al.* and (ii) supports the comprehensiveness of our taxonomy. As reported in Table 5.7, our taxonomy features 16 inner categories that are not covered in [208] (*e.g., To Ensure Better Mapping Between Test And Production Code*), 3 inner categories that are only partially covered (*e.g., Preparing Code for Changes*), and 6 inner categories (*e.g., Forster Code Reuse*) that are completely covered. We provide in our replication package [34] a spreadsheet reporting the mapping between the two taxonomies.

In the following, we discuss each root category, reporting interesting examples and outlining implications for researchers and practitioners, as well as highlighting the differences with the taxonomy provided by Silva *et al.* [208]. The complete list of manually analyzed PRs together with their refactorings/assigned tags is publicly available [34].

Improve Code Design (425 instances). Unsurprisingly, a large proportion of the analyzed refactorings are aimed at improving code design from several perspectives [97]. In 58 of these, the refactorings are aimed at making source code easier to be reused (see *Foster code reuse* in Figure 5.2). In 42% of these cases, this was accomplished through a combination of several refactoring operations, while in the remaining 58% specific refactorings were applied in isolation. When

Table 5.7: Comparison with refactoring motivations found by Silva *et al.* [208]

↑ ↑ highlights a perfect match (the motivation also emerges from their study); ↑ highlights a partial match (we found additional, specific motivations); ↓ ↓ stands for a mismatch (the motivation was not found in their study).

Root Category	**Inner Category**	**Match**
Improve Code Design	Foster code reuse	↑ ↑
	Improve inheritance	↑
	Improve encapsulation	↓ ↓
	Improve maintainability	↑ ↑
Improve Understandability & Readability	Improve naming	↑
	Cleanup code	↓ ↓
Improve Quality of Test Code	To ensure better mapping between test and production code	↓ ↓
	To simplify testing activities	↑ ↑
	To automate tests	↓ ↓
	To remove flaky tests	↓ ↓
	To ease locating tests	↓ ↓
	To simplify testing for client projects	↓ ↓
Preparing Code for Changes	For implementing a new feature	↑ ↑
	For going open source	↓ ↓
	For future refactoring/major change	↓ ↓
	Refactor modules to prepare for next release	↓ ↓
Prevent Bugs	Promote type safety	↓ ↓
	Improve exception handling	↓ ↓
Other Motivations	To create separate Maven artifacts	↓ ↓
	To promote API compatibility	↑ ↑
	To support third-party tools	↓ ↓
	To simplify API usage	↓ ↓
	To allow serialization	↓ ↓
	To improve performance	↑ ↑

this happened, almost always (91%) an operation aimed at extracting a code component from an existing one was applied. In particular, *extract method* operations were performed in 70% of cases to extract a small piece of functionality from an existing method thus avoiding code duplications and fostering the reuse of the extracted code. For example, during the code review of the PR#626 in the `nakadi` project [24], the reviewer observed that two of the implemented methods were "*almost the same except the very last line*" and suggested to "*extract a helper method*" in such a way to reduce code duplication and also allow other methods, in future, to reuse the same functionality. This was accomplished through an extract method refactoring. In other cases, the refactoring was more substantial and directly justified by the need for reusing specific pieces of functionality, as discussed in the PR#488 of the `dropwizard` project [6]. Here the contributor explains, when submitting the PR: "*I was looking at starting/stopping a Dropwizard app in Cucumber tests and DropwizardAppRule has all the functionality I need but obviously it doesn't expose startIfRequired and stop methods. I'd happy to extract a* `DropWizardAppTestSupport` *class from* `DropwizardAppRule`". After approval, this triggered an *extract class* refactoring. This last example is interesting for several reasons. First, extract class is a non-trivial refactoring possibly having substantial ripple effects in the system, with the obvious possibility of introducing bugs. For example, the discussed commit impacted a total of 499 lines of code, thus showing that code reuse is a strong motivation for triggering refactoring operations. Second, while many approaches to identify extract class refactoring opportunities have been proposed (see *e.g.,* [95, 45]), they focus on the identification of complex classes implementing several responsibilities (*i.e., God* or *Blob* classes [56]) that could be split into several classes. The class subject of the *extract class* refactoring (*i.e.,* `DropwizardAppRule`) is a fairly simple class composed by 156 effective LOC (excluding comments and blank lines) that is unlikely to be reported by refactoring recommenders as a candidate for *extract class* refactoring. To cope with these cases, these recommenders could be combined with clone detectors [194] to factor out a class to be used by multiple other ones. Note that these "special" cases should complement the more standard refactoring recommendations done for complex and low-cohesive classes. Indeed, as shown in our RQ_1, classes characterized by a low cohesion as assessed by the C3 and HsLCOM metrics are more likely to be subject to refactoring operations.

Many (338) of the refactorings performed in the code design taxonomy aimed at *Improve Maintainability* (see Figure 5.2). In this category, refactorings aimed at improving the modularization were often implemented through simple move class refactorings, while we rarely observed massive package reorganizations (7 cases). This is in line with recommendations from previous literature, suggesting that approaches performing big-bang remodularization through clustering algorithms have limited applicability, and techniques suggesting fine-grained and incremental adjustments to software modularization should be preferred [106, 165]. Also, it was interesting to find out why developers decided to perform remodularization. For example, in some cases move class refactorings are performed to group, in specific "API-related" packages, utility classes potentially useful in different parts of the system and/or to third-party components, *e.g.,* PR#324 from `DSpace` "*I suggest to move this class in dspace-api as it will be useful to port this feature to JSP UI as well*" [8]. While these changes might look suboptimal from the cohesion-coupling point of view (*i.e.,* they could generate a low-cohesive package) they are justified by a clear rationale. As also observed for the approaches automating *extract class* refactoring, tools recommending modularization solutions (see *e.g.,* [35, 127, 143, 183]) just strive to maximize the cohesion-coupling trade-off. Given the availability of historical data, they could also learn from previous changes what a meaningful modularization is from the developers' perspective. While learning code changes is already an active research field [225], no previous work has attempted to design refactoring recommenders learning from developers' activities what a meaningful refactoring is in a given context.

Removal of code clones is one of the two motivations behind the refactorings in the *Adhere to DRY principle* (Don't Repeat Yourself) subcategory (child of *Remove Code Smell*), together with the removal of unnecessary code. Note that this category is strictly related to the *Foster code reuse* one. Indeed, some of the analyzed PRs fall into both these categories because factoring out duplicates also creates a more generic code element (*e.g.,* a class or method) that can be further reused. For example, PR#366 in the `fineract` project [2] can be seen as an example of improving reusability by adhering to the DRY principle, since it features an *extract method* refactoring suggested by the reviewer and avoiding code duplication while allowing the reuse of a piece of functionality now embedded in the extracted method. This confirms once more the relevance for practitioners of clone detectors [194] as well as of refactoring tools aimed at removing clones [223] and encourages their use in the Continuous Integration (CI) pipeline, as advocated by Duvall *et al.* [81].

Concerning the removal of unnecessary code, besides cases simply related to removing unused `imports`, we found refactorings performed to remove redundant code (*e.g.,* PR#1481 from `testng` [3]). While some work has investigated the automatic identification of redundant code in software systems [133, 118], the provided support is still very limited to specific redundancy cases (*e.g.,* those related to API usages [118]) or programming languages (*e.g.,* LISP [133]). The 55 cases related to refactorings motivated by the removal of unnecessary code suggest room for more research in this field.

Concerning the operations targeting a better distribution of the responsibilities across code components, one very interesting example comes from the `DSpace` project (PR#1083 [7]). This PR implements a massive refactoring aimed at ensuring a better "separation of concerns/responsibilities" for an API module, and has been subject to votes by the community, because the refactored API was not backward compatible. Despite this issue, the merging has been approved thanks to the numerous positive advantages brought by the refactored API: "*makes it much easier to achieve future goals on our Roadmap, especially, moving us towards potentially better support of third-party modules*", "*it cleans up one of the messiest areas of our existing API [. . .]*". This case shows the non-trivial trade-offs that developers should consider in case of massive refactoring: for example, smartphones have limited battery life and they require software optimized to reduce the energy consumption. State-of-the-art refactoring recommenders [221, 45] ignore the heterogeneity of modern software, and the different priorities that non-functional requirements, possibly more important (*e.g.,* maintainability, performance, backward API compatibility) may have in different contexts. Future work should consider integrating into these recommender systems the possibility to define a priority list of non-functional properties that developers are or are not willing to sacrifice when applying refactoring. This would allow generating more meaningful and sensible refactoring recommendations.

The two sub-categories described above (*Foster code reuse* and *Improve maintainability*), *i.e.,* the ones highly represented in our study, have a complete matching with the motivations identified by Silva *et al.* [208]. This confirms their findings, and stresses once more the importance from the developers' perspective of improving both the reusability and maintainability of code, especially when discussing whether to accept or not a PR.

Other less represented subcategories in the *Improve code design* taxonomy include refactorings aimed at improving the usage of inheritance (21 instances) and the ones working on the encapsulation (5). In these cases, considering the low number of instances belonging to each category, it is quite obvious that while comparing with the motivations by Silva *et al.* we found that these reasons did not emerge from their study. The only exception is the one related to inheritance that is only partially covered, as shown in Table 5.7.

Improve Understandability & Readability (468 instances). The majority of refactorings we found in the manually-analyzed PRs aim at improving understandability and readability of source code. This supports the findings of RQ$_1$, which indicate a significant correlation (and high OR) of readability metrics with refactoring operations. In this category, 146 refactorings were done to improve naming. The observed renamings had a variety of motivations (see Figure 5.2), ranging from fixing typos to keeping naming consistency throughout the project. Naming decisions were often carefully discussed, showing their importance for developers. An interesting example of discussion about naming is the PR#150 of the `optaplanner` project [11]. The original intent of the PR was to add a new feature, but the author explicitly asked for feedback about the naming of a new interface he extracted: "*we should discuss the naming and the usage of the* `SolverProblemBenchmarkResult` *interface*". Such a name was changed after the discussion: "*renamed to* `BenchmarkResult` *as agreed in a meeting*". In the same PR the developers also discussed several other names in the contributed code. For example, the PR author introduced a boolean field named `hasNonDefaultSubSingleCount`; another developer asked why such a value was introduced, since the name was not clear enough. The discussion triggered not only a renaming operation, but also a type change (from boolean to integer) to represent additional information that could be useful in future: "`maximumSubSingleCount` *is the best name here, as it gives us more potential information for the future at no cost*".

We also found cases of renaming aimed at better reflecting the code responsibility (38). A representative example is in PR#251 of the `kafka-connect-elasticsearch` project [5], in which one of the reviewers suggested to rename a test method to something very specific and clearly depicting the responsibility of the test case: "*you could change the name of the test method to something like* `testCreateAndWriteToIndexForTopicWithUppercaseCharacters`. *I like test names that read like the condition they are testing*". Other cases aimed at removing linguistic antipatterns defined in the literature by Arnaoudova *et al.* [38] and known to have negative effects on the understandability of code [87]. This is only one of the studies linking the poor quality of identifiers to difficulties experienced by developers in code comprehension [217, 131, 130, 76, 129]. Our findings show that developers care about the quality of identifiers and carefully discuss their choice.

Most of the rename refactoring recommendation approaches aim at fostering the usage of consistent naming [28, 139], while only a single attempt has been done, to the best of our knowledge, to recommend rename method refactorings with the goal of better reflecting the responsibilities implemented by the code [32]. Our results show that more effort in this direction is needed since this is the scenario in which developers more frequently perform rename refactorings. Also, the high number of rename refactorings implemented as a consequence of code review indicates the possibility to mine this data to evaluate automated rename refactoring techniques [28, 139]: the originally submitted identifier represents an opportunity for rename refactoring while the one adopted after the code review process can be used as reference of a good refactoring. This would avoid the evaluation of the rename refactoring techniques in artificial scenarios.

Looking at Table 5.7, and considering that developers can modify the names of packages, classes, variables or methods for different reasons, we can state that our *Improve Naming* category is only partially covered by the motivations reported in the previous study by Silva *et al.* [208]. For instance, while both studies identify the need for adhering to naming conventions, for keeping consistency in naming, or for better representing code responsibilities, in our taxonomy we also found cases where the renaming occurs to fix typos, shorten identifiers and expand abbreviations. The latter challenge (*i.e.,* expansion of abbreviations in code identifiers) has been vastly investigated in the software engineering research literature (see, for example, the works by Lawrie *et al.* [128]), with approaches proposed and empirically evaluated. However, to the best of our knowledge, there are no ready-to-use tools that, for example, can be integrated in a CI pipeline and can recommend to developers identifiers to expand at commit time. The implementation of such a tool is a clear next step to perform in this research field.

We also found several refactorings implemented to make the source code less confusing. Such changes involved improvements to both names and structural aspects. For example, we found an interesting example in PR#599 of the `htsjdk` project [19]. Note that the refactoring performed in this PR is an *extract interface* rather than a rename, that resulted in the interface `CRAMReferenceSource` implemented by the class `ReferenceSource`. The main goal of the refactoring was to improve code reusability: for this reason, we include such a case in our taxonomy under the *Foster code reuse* category. However, it is interesting to discuss this refactoring in the context of renaming: during the code review process, one of the reviewers argued that the chosen names were confusing because he expected an inheritance relationship in the opposite direction (*i.e.,* `CRAMReferenceSource` implements `ReferenceSource`) by reading the names alone. As a consequence, `ReferenceSource` was renamed to `CRAMReferenceSourceImpl`, making the relationship between the two classes more evident. This naming issue could be characterized as a sort of linguistic antipattern, and shows that the original catalog of these antipatterns defined by Arnaoudova *et al.* [38] could be expanded by analyzing recommendations provided by reviewers in a code review process.

Finally, we found many cases in which the developers made more generic cleanups in the code (162 instances), to improve the coding style and, in some cases, the quality of error messages and logging. Some of these changes were performed as a result of tools' recommendations. For example, the PR#346 of the `spring-amqp` project [22] fixes warnings raised by SonarQube. This suggests that developers are willing to fix issues identified by automatic tools. However, in our sample of PRs, SonarQube was the only tool mentioned in many discussions. Note that a specific analysis of the extent to which static analysis tool warnings are removed was not in scope of our work (but rather addressed in related literature [73, 121, 210]); this is the reason why, in RQ_1, we only considered two tools — DECOR and PMD — that could raise warnings that triggered refactoring operations.

More than 60% of the clean-up operations were done as part of the original intent of the PRs. Differently from other categories, we found very little discussion among developers regarding clean-ups. This is likely due to (i) the limited impact that these clean-ups generally have; and (ii) a general agreement on the need for improving code quality.

For these instances, we analyzed the values of readability metrics pre and post refactoring operations made to improve understandability and readability. We found that the readability of such file – measured with the suite of metrics provided by Scalabrino *et al.* [199] – improves in a small minority of cases. Indeed, the readability improves in 32.1% of files and worsens in 38.4% of files. Furthermore, files maintain their readability level in 29.5% of the total of files. We obtained a similar trend also with the metrics by Buse and Weimer [59]. Specifically, the readability improves in 8.3% of files and worsens in 12.3% of files. In addition, the readability remains stable in 79.4% of files.

Prevent Bugs (26 instances). These refactorings are motivated by the will to prevent bugs, for example through a better exception handling or by promoting type safety. Note that these are changes that preserve the program's behavior. For this reason, we contemplated them in our taxonomy, even if they do not belong to the canonical cases of refactorings, such as those defined by Fowler [97]). This is why those categories are completely uncovered in the 44 motivations provided by Silva *et al.* [208]. Indeed, they studied the reasons behind specific refactoring operations that are detected by RMiner and inline with those defined by Fowler [97].

Some PRs are explicitly motivated by the will of improving the exception handling mechanism. This is the case for PR#933 of the `nakadi` project [25], in which the developer implements several different refactorings (*e.g., rename class*, *move class*) to improve the overall handling of the exceptions in the project. Other PRs, instead, simplify the handling of exceptional conditions. For example, in PR#1067 from the `htsjdk` project [18], the developer fixes a possible issue caused by the invocation of the method `mFile.getSource()` within several exception messages. Indeed, in specific cases the mFile object could be null, leading to the throwing of a NullPointerException. For this reason, the developer implemented an *extract method* refactoring, creating the method getSource() which returns the value of mFile.getSource() when mFile is not null, and a constant string otherwise. This allowed to easily prevent NullPointerException by replacing the many usages of mFile.getSource() with an invocation to the newly created getSource() method. This is an interesting application of *extract method* refactoring, since it aims at refactoring a very small clone, *i.e.,* a method invocation reused, in the same way, in different parts of the code. Extract method is widely applied in the refactoring of clones [125]. However, the focus is usually on more complex clones sharing several statements, rather than on the identification of refactoring opportunities that, as in the discussed case, involve few code tokens but can have a positive impact on the reliability and maintainability of the system.

Another very interesting example is the PR#238 from the minio-java project [16]. Here the developer replaced general, unchecked exceptions such as NullPointerException, with more specific and checked ones. With "unchecked" we refer to those exceptions that in Java can be thrown without declaring them in the method signature. For example, in the specific case of PR#238, the method getClient of the class Client was throwing a NullPointerException in specific situations: note that this was an intended behavior of the method, *i.e.,* there was an explicit throw new NullPointerException() in the code. However, in the method signature the only visible exception was MalformedURLException. Through the implemented changes, including *class rename* refactoring, the more general exceptions have been specialized (*e.g.,* to ClientException in the case of the getClient method), forcing the exposure in the signature of the thrown exception. This has the double effect of (i) giving developers compilation errors if they do not catch the thrown exception, thus preventing bugs; and (ii) using more expressive exception names. Note that the usage of unchecked exceptions in Java code should not be considered as a "bad smell" since, in general, unchecked exceptions should be used to reveal bugs, while checked exceptions to throw errors that the program should handle [68]. However, the misuse of unchecked exceptions where checked ones are needed can lead to higher chances of introducing bugs (as in the case of PR#238). The automatic identification of these situations is, to the best of our knowledge, a problem still not faced in the research literature.

Preparing Code for Changes (41 instances). This category includes refactorings facilitating the implementation of new features or of other planned activities. Refactorings preparing for future changes are usually implemented in dedicated PRs including major refactorings.

Looking at the comparison highlighted in Table 5.7, it is possible to state that our taxonomy provides more insights compared to the list of motivations in the previous study. Indeed, the category *Preparing Code For Changes* contains some motivations already highlighted in [208] while others missed such as the need for refactoring operations aimed at moving to open-source. An interesting case in this category is represented by PR#317 of the `sagan` project [21]. As mentioned in the title, the goal of the PR is to *"refactor in preparation for open source"*. Note that this is kind of an exception in our taxonomy, and we decided to put it into the *Preparing Code for Changes* category just because open sourcing a project is, in some way, a decision taken to foster the future evolution of the project. Code refactoring is one of the action items in a checklist defined in issue #179 for preparing the project to be open-sourced [20]; other action items included, for example, the introduction of installation and configuration instructions. This suggests that an appropriate code cleanup, including refactoring, should be part of packaging checklists when putting a project in the open-source.

Another example of refactoring performed to accommodate other changes is in PR#136 of the `zhcet-web` project [26], where the developer extracted the class `CryptoUtils` from `SecurityUtils` to accommodate the implementation of new functionalities (*e.g.,* the `decrypt` method) in a suitable class (*i.e.,*`CryptoUtils`).

Tools supporting preemptive refactoring are lacking in the literature. Indeed, the only effort in this direction is the work by Pantiuchina *et al.* [169] in which, however, the focus is on identifying classes that will be affected by code smells in the future, thus recommending them for a preemptive refactoring action. Our manual analysis indicates that a novel family of recommender systems able to suggest developers how to refactor the code in order to "accommodate" the implementation of a given change request could be valuable.

Improve Quality of Test Code (49 instances). As the production code might be in need for refactoring, this also holds for test code [227]. The quality of the test code is also assessed in the context of PR discussion and code reviews, as observed by Spadini *et al.* [211]. We found 49 test code refactoring cases, 37% of which performed with a specific type of refactoring operation, and 63% with multiple operations. The observed refactorings include changes similar to those performed on production code, *e.g.,* better distribution of responsibilities to have a better mapping between test and production code, see *e.g.,* PR#1071 in the `error-prone` project [10] in which the author comments "*[...] separate the tests into logical classes*".

Other cases we found concern the removal of flaky tests which introduce non-determinism in test outcome [149]. In the `microprofile-fault-tolerance` project, the PR#363 [9] aims at removing flaky test: "*The tests* `testCircuitInitialSuccessDefaultSuccessThreshold` *and* `testCircuitLateSuccessDefaultSuccessThreshold` *were moved to an independent test to avoid dependencies between tests that use the same bean [...] that can generate possible failures when the circuit breaker leaves open [...]*".

More interesting and specific for test code are the refactorings performed to improve testability. In PR#73 of the `WPS` project [1] a developer performed an *extract class* refactoring in production code motivated by the will of simplifying integration testing: "*The functionality to create a GTVectorDatabinding out of shapefiles was removed from the* `GenericFileData` *class and moved to a new* `GenericFileDataWithGT` *class. Due to this change, the processes used for the integration tests do not depend on GeoTools anymore [...] also the tests now use only local resources*". This example shows how refactoring performed on production code can have an impact on many software quality aspects (in this case, testability). Cases like this one suggest to always ponder the positive and negative impacts of refactoring beyond maintainability *e.g.,* some refactoring actions can aid testability, while others might improve maintainability at the cost of testability. Clearly, considering this aspect in the context of a refactoring recommender is far from trivial, given the need for automatically assess the testability of a given component. Besides, the presence of refactorings specifically aimed at improving testability shows room for approaches aimed at recommending such kind of operations.

While the motivations of refactorings for simplifying testing activities were already presented in the work by Silva *et al.* [208], our taxonomy provides other new categories such as those aimed at removing flakiness, at automating testing activities, or at improving the overall testability of the project under development.

Other Motivations (108 instances). In this category, we put all motivations that did not find their place among other root categories.

Thirty-seven PRs were performed to improve software performance. This is not surprising and in line with Fowler [97], who observed that the internal program structure is closely related to its performance due to better optimization opportunities. Moreover, the latter also emerges from the motivation provided in the previous study by Silva *et al.* [208]. Also, our quantitative results of RQ_1 indicate that refactorings have a high chance to occur on classes frequently subject to bug fixes, which may have affected performance, especially in the case of quick patches. A concrete example is the PR#1577 of `AmazeFileManage` [23] Android application. In a linked issue, a user reports that when she is "*copying a large file using SFTP, the process can take more than 1 minute, so the phone goes in stand by mode*". PR#1577 improves the I/O performance of this feature through refactoring.

This example confirms the importance of specific non-functional attributes (in this case, performance) for different types of software (in this case, a mobile app). Also, once again, it points to the need for developing refactoring techniques able to consider this heterogeneity of non-functional requirements rather than mainly focusing on maintainability as done in state-of-the-art refactoring tools [221, 45]. Indeed, to the best of our knowledge, only a few authors have developed refactoring techniques having the improvement of performance as the main objective [79, 243, 37]; however, these approaches either target very specific performance issues [79, 243] or are designed to work on models rather than on source code [37].

Link between qualitative and quantitative reasons.

Quantitative and qualitative reasons pushing for refactoring match each other. Indeed, if we compare metrics with qualitative reasons, it is interesting to note that each metric matches at least one of the qualitative reasons. For example,

- *LackStructRead* and *LackComRead* match with *Improve Understandability & Readability*;
- *LackC3*, *NOM* and *HsLCOM* match with *Improve Maintainability.* Specifically, the first two metrics match with *Better distribute responsibilities* of *Improve Maintainability*;
- *NOPM* matches *Improve Maintainability.*

The same considerations hold also for process metrics. In particular,

- *FilesRelatedToIssueFix* and *AvgLinesImpactedInCommit* match with *Prevent Bugs*;
- *DistanceNextRelease* and *DistancePreviousClasses* matches with *Preparing Code for Changes.*

We also observe that code design flaws metrics, such as *IsGodDecor*, *IsCDSBPDecor*, *IsComplexDecor*, *IsFuncDecDecor*, *IsSpaghCodeDecor*, match with *Improve Maintainability*, while *ExcessiveParameterList* matches with *Improve Inheritance* and *TooManyList* and *TooManyMethods* match with *Improve Understandability & Readability.* Finally, the remaining quality metrics match with *Improve inheritance* (*i.e., WMC*, *DIT*, *NOC*, and *RFC*), with *Improve maintainability* (*i.e., CBO* and *NOSM*), with *Improve Encapsulation* (*i.e., NOF*, *NOSF*, and *NOSI*), and with *Foster Code Reuse* (*i.e., LOC*).

We conclude RQ_2 stating that quantitative and qualitative reasons pushing for refactoring match each other. In the case of low-cohesive classes, developers need to increase the cohesion of those class. Thus, once that developer performs the relative change, the related metrics were undergoing a positive change. This improvement has been to analyze also in the readability metrics.

5.4 Threats to Validity

Construct validity. A source of inaccuracy is represented by the automated refactoring detection. However, RMiner has been reported to exhibit a very high precision (98%) and recall (87%) [222]. This threat is mitigated at least in RQ_2, where refactorings have been manually reviewed. To identify bug fixes, we used an approach matching regular expressions onto commit messages [91], as also done in previous work [189]. To limit threats due to this heuristic [36], two authors independently analyzed, for each project we considered, a random sample of commits classified as a bug fix to mark true and false positives. After discussing disagreements, only 8% of the analyzed commits resulted to be false positive bug fixes (mostly related to CheckStyle fixes).

In RQ_1, we only analyzed the correlation between the presence of any refactoring with various metrics. While it may be interesting correlating specific types of refactorings with metrics, our qualitative analysis showed that refactoring goals are often achieved through a combination of refactorings. To build the explanatory model of RQ_1, we have selected a broad set of metrics capturing different aspects of software product and process. It is important to note that the aim was to correlate such metrics with the presence of at least one refactoring action of any kind. Building models for specific refactoring types is out of scope of this study and could, possibly, require to identify further specific indicators.

In RQ_2, we identified refactoring-related PRs as those having (i) one of their commits containing a refactoring identified by RMiner, and (ii) a refactoring-related keyword in their title. Such selection criteria can result in false negatives (*i.e.,* missing some refactoring-related PRs) and, in turn, this may have resulted in missing categories in our taxonomy. Indeed, it is possible that our taxonomy is only representative of the motivations behind PRs that can be captured through the adopted selection criteria.

As context for our study we targeted non-personal/toy projects having a substantial change history to study and being active. For this reason, we defined a number of selection criteria (*i.e.,* at least 5 contributors, 1 fork, 500 commits, 100 PRs, and one recent commit) that, however, may fail in capturing the type of systems we were interested in.

Internal validity. In the quantitative analysis (RQ_1), although we tried to capture factors from different dimensions (*i.e.,* different kinds product and process metrics), there could be many other factors that could have influenced the need for refactoring. We mitigated this threat through (i) the use of a mixed-model considering project as random effect, and (ii) the qualitative analysis of RQ_2. It is important to note that the aim of RQ_1 is to mainly identify correlations between metrics and refactoring activities, and not about claiming any causation. Only the qualitative analysis of RQ_2, taking into account developers' discussion, can highlight the rationale for refactoring actions.

Conclusion validity. In RQ_1, we performed a careful preprocessing of data and variable selection to avoid multi-collinearity, and normalized metrics to allow properly interpreting the ORs.

External validity. Our analysis is limited to a sample of 150 Java open source projects hosted on GitHub, and the qualitative analysis to 551 PRs. We do not claim the generalizability of our findings to other programming languages or to industrial systems. For this reason, a further investigation on a more diverse set of projects, developed with different programming languages and belonging to both open and closed-source is highly desirable. Also, it is worth mentioning that in our manual analysis we only considered PRs for which RMiner identified at least one refactoring operation. This means that we did not consider PRs that, for example, targeted a complete remodularization of the system that involved refactoring operations not captured by RMiner.

5.5 Final Remarks

The goal of this study is to analyze the motivations behind refactoring operations, observing code and real discussions of developers. In this study we quantitatively and qualitatively analyzed the reasons behind refactoring operations performed by developers. Our quantitative analysis highlighted that (i) code readability is the product-related factor mostly correlated with refactoring operations, and (ii) process-related factors such as source code change- and fault-proneness and, especially, the experience of developers changing a code component, play a significant role in triggering refactoring operations. Our qualitative analysis resulted in an extensive taxonomy of 67 motivations behind refactorings, relating to quantitative results where possible. We have made the study material and data available in our replication package [34].

From results of our study, we can derive the following lessons learned:

- Developers improve code design in multiple classes, but this can create problems in many cases. For example, extract class refactoring is a non-trivial refactoring that can take to ripple effect and multiple bugs. However, this operation can maximize cohesion and coupling.

- We confirm previous studies, *i.e.,* fine-grained techniques and incremental adjustments are preferable to approaches of big-bang remodularization.

- Clone detectors help to remove clones and to promote the use of the Continuous Integration (CI) pipeline. Regard the removal of unnecessary code, the support is very limited for some redundancy cases (*e.g.,* API usages) or programming languages (*e.g.,* LISP).

- In case of massive refactoring, developers should consider the heterogeneity of modern software and the different priorities of non-functional requirements in different contexts.

- Developers care about the quality of identifiers. In addition, rename refactoring recommendation approaches suggested a coherent name, which can reflect variable responsibilities implemented by the code. Rename refactoring are often associated to a code review and this allows to create an automated rename refactoring technique.

- Developers find useful fix issues identified by automatic tools (*e.g.,* SonarQube).

- *Extract method* refactoring is a valid technique to reuse methods in different parts of the code.

- Developers recommend appropriate code cleanup and refactoring when the project is open-source.

- Refactoring of production code can impact on different software quality aspects (*e.g.,* testability).

CHAPTER 6

Do Developers Care about Code Readability?

Contents

6.1 Introduction

Software refactoring is performed to improve code readability metrics. Indeed, developers declare to do refactoring to improve code readability, as previously shown in Chapter 5 [171]. This result confirms previous searches: *"readability"* is a fundamental and highly desirable property of the source code.

Code readability is the very first step during incremental change [51, 186]. This activity is required to perform concept location, impact analysis and the corresponding change implementation/propagation. Assuring source code readability becomes imperative in modern open-source software development due to its collaborative and geographically distributed character [51, 186]. Erlikh [86] has shown that during software evolution tasks developers spend plenty of time maintaining the existing code (often written by others), far more than writing code from scratch. Indeed, when developers have to understand an unfamiliar code, they invest 35% of their time to navigate it because they need to collect information on code [123]. Even if sometimes expert developers are more able to understand a code because they use recurring and structured comprehension strategies [192].

From the achieved results of the previous study (Chapter 5), we can deduce that any developer would prefer working on readable code rather than on unreadable code. This deduction is the result of their willingness to do refactoring. It is less evident, instead, to what extent developers think that making an effort to keep the source code readable is important and worthwhile during software evolution. Previous studies investigated the developers' perception of code readability [170, 88]: such studies provide *implicit* evidence that code readability matters to developers since some commit messages mention the tentative improvement of code readability, it can be deduced that developers strive to make the code more readable. To the best of our knowledge, no previous work tried to look at *explicit* evidence that developers care about code readability in software evolution.

For this reason, in our study, we wanted to understand what is the developers' perception of code readability. Thus, we conducted a survey with 122 developers to verify to what extent code readability is important to them while writing or reviewing code. To do this, we constructed a survey composed by four questions, which could be answered using a 5-point Likert Scale [136]. After, we distributed the survey on *general purpose* and *specific purpose* social networks. Finally, we invited other possible participants through apposite invitations. As result, we found that the vast majority (∼83.8%) often take code readability in account when writing code.

The remainder of this chapter is organized as follows. In Section 6.2 we report the design of our study and in Section 6.3 we report the results of our study.

Finally, Section 6.4 discusses the threats to validity of the studies and Section 6.5 concludes this chapter.

6.2 Design of the Study

The *goal* of our study is to understand the perception of developers about code readability, *i.e.,* to what extent they consider readability important while writing or reviewing code, and if they actively modify the code to make it more readable.

With our study we want to answer the following research question: *What is the developers' perception of code readability?* To answer such a research question, we surveyed software developers. The survey consisted in four main questions:

Q_1 *When you write code, to what extent do you take into account code readability?* We ask this question to understand if developers think about code readability while producing new code.

Q_2 *When reviewing code changes performed by your peers, to what extent do you consider the impact of the change on code readability?* We ask this question to understand if developers think about readability when discussing about the approval of a change.

Q_3 *How often do you make changes to improve code readability?* With this question we want to understand if developers sometimes pause their activities specifically to improve the readability of previously written code.

Q_4 *How frequently did you experience a big change in code readability (positive or negative) in the projects you worked on?* With this final question we want to understand what is the perceived frequency of readability changes. We later compare the answers to this question to the results we obtain in our subsequent study (Section 7.4).

The first three questions (Q_1, Q_2, and Q_3) could be answered using a 5-point Likert scale, ranging from 1 (*Never*) to 5 (*Always*). Specifically, we used the Likert scale [136] because it is usually used to measure the level of agreement or disagreement on a symmetric agree-disagree scale for a series of statements.

For the last question (Q_4) we used a different 5-point Likert scale: the developers could choose one of the following options: (i) "*Never*", (ii) "*For less than 25% of the changes*", (iii) "*For more than 25% and less than 50% of the changes*", (iv) "*For more than 50% and less than 75% of the changes*", (v) "*For more than 75% of the changes*".

Besides, we asked demographic questions, *i.e.,* (i) education level, (ii) occupation, (iii) experience compared to the colleagues (following the recommendation by [206]), (iv) the three most used programming languages, and (v) the number of contributions in open source and industrial projects.

We distributed the survey on social networks. We have chosen both *general purpose* social networks, *i.e.,* Facebook, Twitter and LinkedIn, and *specific purpose* social networks, *i.e.,* Reddit. For this last channel, we posted the invitation on two sub-reddits in which surveys are allowed, *i.e.,* r/SoftwareEngineering and r/SampleSize.

Furthermore, we personally invited other possible participants (*i.e.,* students and developers in software companies).

Finally, we try to understand if there is a correlation between when developers peer-review the code written by others and when developers write their code. To do this, we report Kendall's τ, along with the p-values.

6.3 Results

We obtained 122 responses, 77 of which by developers personally invited by the authors (63.1%), 23 by Reddit users, 11 by Twitter users, and 11 by Facebook users.

In Figure 6.1, we show the distribution of answers to the demographic questions we asked. More than a 50% of the surveyed developers work in industry (65, *i.e.,* 53.3%), 28 work in Academia (23.0%), while 42 of them are students (*i.e.,* 34.4%). It is worth noting that developers could select also more than an option for occupation (*e.g.,* they could select both "Working in industry" and "Student").

As for the education, 21 participants have a PhD (17.2%), 51 have a master's degree (41.8%), 36 have a bachelor's degree (29.5%), a small part of them have high school degree (14, *i.e.,* 11.5%) instead. The five most popular programming

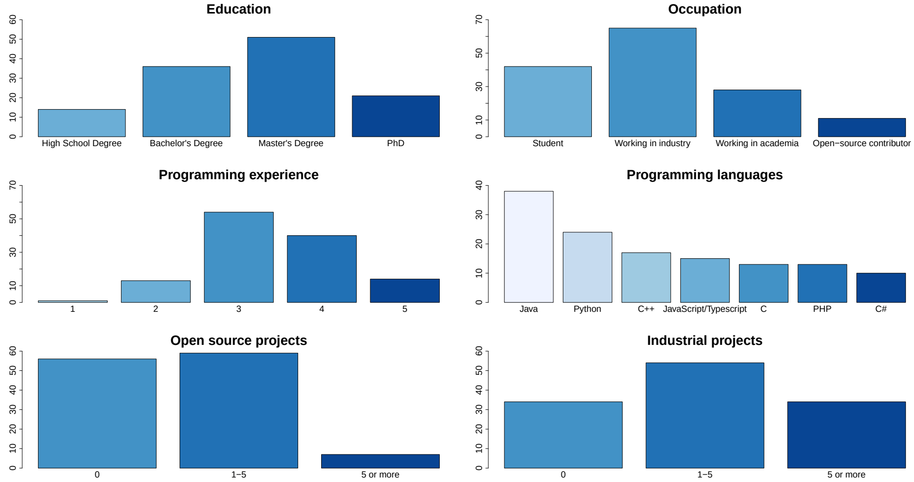


Figure 6.1: Distribution of the answer to demographic questions.

languages commonly used by the developers are Java, Python, C++, JavaScript or TypeScript, C. Most developers involved in the survey stated that they have a similar programming experience compared to their colleagues (54, *i.e.,* 44.3%); the same amount of developers (54, *i.e.,* 44.3%) think that they are more experienced (40) or much more experienced (14) than their colleagues. Finally, only a small portion of developers said that they are less (13) or much less (1) experienced than their colleagues (11.4%).

We report in Figures 6.2 and 6.3 the distribution of the answers to the four questions. We use two different colors to highlight the differences between developers who did not contribute to any collaborative project and developers who contributed to at least an open source/industrial project. Figure 6.2 shows the distribution of the answers to Q_1, Q_2, and Q_3. About 44.6% of the participants **always** take into account code readability when writing source code, while 41.7% of them **often** consider it. We obtained similar results also for Q_2: 34.0% of developers say that they **always** consider readability while peer reviewing code and 46.6% of them **often** take it into account. The same trend is visible for the answer to Q_3: 40.8% of developers say that they **often** perform changes to improve code readability and 31.1% of them **sometimes** improve it.

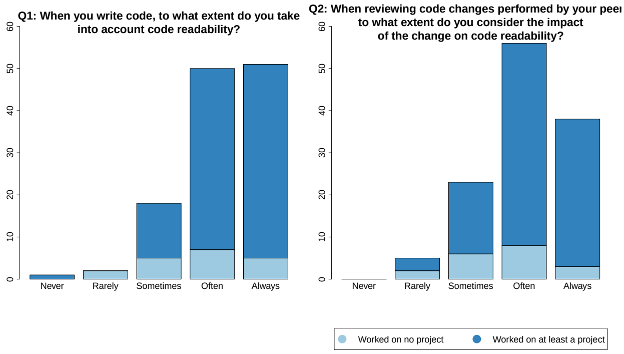


Figure 6.2: Distribution of the answers to Q_1, Q_2, and Q_3.

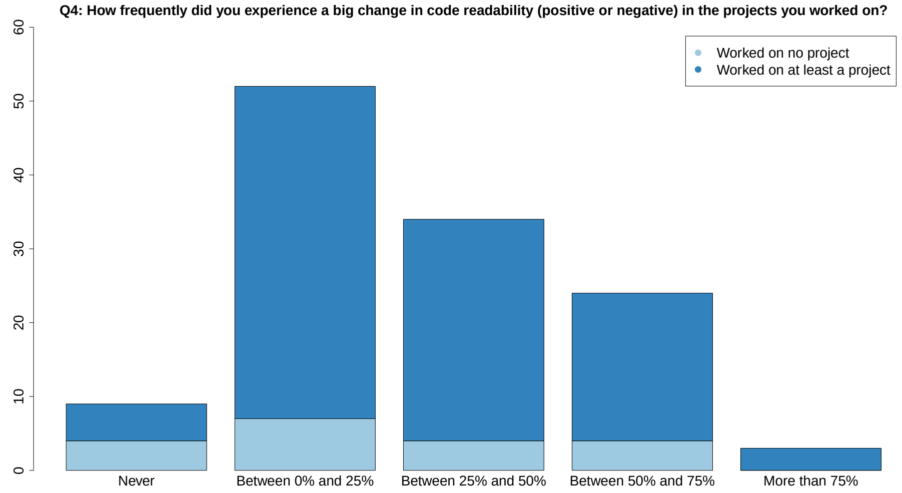


Figure 6.3: Distribution of the answers to Q_4.

We found that 26.2% of the participants stated that they consider readability more when writing code than when reviewing code written by their peers, while the opposite happens only in 13.1% of the cases. We statistically analyze the difference between Q_1 and Q_2 to verify whether developers give different importance to readability when peer-reviewing code written by others than when they write their own code. We used the Wilcoxon rank-sum test [236] to check such hypotheses. The difference is not significant (p-value $= 0.07$) and the effect size is negligible. This suggests that developers understand the importance of readability, giving a similar priority to code writing and peer-reviewing code. This can have many causes, *e.g.,* it could happen for social reasons [168].

Finally, Figure 6.3 shows that many developers perceive that big changes in code readability are not very frequent (less than a quarter of the files for 52 developers). Considering both Figures 6.2 and 6.3, it is interesting to notice that the contributions to projects do not seem to affect the answers given by the developers.

Such results show that readability is very important to developers since they (i) take it into account while writing code, (ii) value it while reviewing code changes, and (iii) make an effort to improve it, when possible. Besides, most of the developers perceive that readability rarely changes: we compare the perceived results with the our empirical results in Section 7.4.

Summary of the motivating study. The large majority of surveyed software developers care about code readability during software developement.

6.4 Threats to Validity

In this section, we discuss the affected threats that could affect the validity of the results.

Construct Validity. One threat to construct validity is the choice of questions for the survey because these cannot be complete and clear. One of the main reasons can be the language because we may have written incorrect questions, given that we are non-native English language. Vice versa, we may have written correct questions, but non-native English language developers could misinterpret questions.

Another threat can be the setting of the survey question (*i.e.,* Q_1, Q_2, and Q_3). These survey questions could take to positive answers over negative answers: a developer would answer she cares about code readability just because every good developer is supposed to care about readability, not because she really cares about it. We tried to mitigate such a threat by making the survey anonymous. However, there is still the risk that developers positively answer to the above questions just to satisfy their self-esteem.

Another threat is the distribution of surveys on social networks. About *specific purpose* social networks (*i.e.,* Reddit), we posted the invitation on two sub-reddits (*i.e.,* r/SoftwareEngineering and r/SampleSize), but we could choose other sub-reddits that include a large set of developers not present in the other chosen two sub-reddits.

Internal Validity. One possible threat to internal validity is the measurement of each single question. We use for each question 5-point Likert Scale, but the Likert Scale could be based on 7-points. Thus, we could give more levels of response to each single question [136].

External Validity. The main threat to the extenal validity is the involvement of a large number of participants. The achievement of a large number of participants is a very difficult goal on professional programmers.

In our study, the big main to the external validity is the achievement of programmers through on different *general purpose* social networks (*e.g.,* Facebook). Thus, we trust that people not belonging to the software engineering world did not participate in this survey and we obtained truthful data.

6.5 Final Remarks

Readability is one of the main activities of a developer. A d eveloper has to read a code to change it. This activity is fundamental when more developers work on the same code, but it is not clear to what extent it is important for a developer to maintain readable code.

In this study, the goal is to understand what developers' perception of code readability is. We constructed a survey of four questions. Each question could be answered through a 5-point Likert scale, where in three question developers could decide their level of agreement or disagreement with the specific question and in one question the developer could choose a specific option.

We involved developers posting an invitation on *general purpose* and *specific purpose* social networks. The former are Facebook, Twitter and Linkedin, and the latter is Reddit. Finally, we invited other possible participants through our network of knowledge.

From our results, we can derive some lessons learned:

- software developers consider important code readability during their software development activities. Indeed, they perform changes to improve code readability.
- software developers should consider important code readability during peer reviewing code. Indeed, developers should consider code readability also when reviewing a code.

From these lessons learned, we try to define a model for describing code readability evolution [175], that in this thesis will be explained in Chapter 7.

CHAPTER 7

Readability Evolution in Open Source Projects

Contents

7.1 Introduction

As previously shown in Chapter 5 and Chapter 6, code readability is important for developers to the point for performing refactoring operations to improve code readability [171, 175]. In addition, in literature, several facets have been reported as components that contribute to making code readable [146, 164, 48]. Such components are usage of design concepts, source code lexicon, and visual aspects (*i.e.,* syntax highlighting). Indeed, previous works provide empirical evidence that structural [58, 181], visual [80], and textual [200, 199] aspects can be used to automatically assess code readability, shedding some light on what makes code readable or unreadable. Such studies tend to focus on a single version of a software artifact. However, software is a palimpsest with subsequent changes applied on top of the previous ones.

This is why one can plausibly expect source code readability to be an outcome of a complex process involving multiple actors and revisions. To the best of our knowledge, the literature provides only few hints on how readability changes, why some parts of the system start to become less readable and what developers do to prevent it. Lee *et al.* [132] explored 210 open source Java projects to study the existing relationship between source code and violated coding convention. They found that code readability is affected only by some of such code violations (such as Javadoc-related ones), while it is not affected by others (*e.g.,* class design-related). Spinellis *et al.* [214] studied the evolution of programming practices in Unix and, among the other aspects, they considered readability using some common readability metrics, such as statement density and comment character density (*i.e.,* comment characters divided by the total number of characters across all source code files). The authors found that readability in Unix has increased over time. Such a study provides some interesting insights on how readability evolves; however, it is focused on a single software system and, therefore, it is not clear if these findings are true also for other systems and other programming languages.

Given the results of previous studies (Chapters 5 and 6), we wanted to understand the dynamics related to the evolution of code readability during the lifecycle of a software system. Specifically, in this study, we wanted to understand

and take a closer look at how code readability changes in the evolution of software systems. Then, we defined a model able to describe the readability evolution of a given file in a software project. This model is a Markov chain based on two main states, *i.e.,* file being *readable* and *unreadable*, and two initial states, *i.e., non-existing* (when the file is not created yet) and *other-name* (when the file exists with a different name). We used the most accurate tool available in the literature [199] to decide if a snapshot of a file was *readable* or *unreadable.* Such a tool, like the other ones in the literature, was designed to work on single code snapshots: it is unclear what is the accuracy achieved in classifying readability transitions (from {*readable*/*unreadable*} to {*readable*/*unreadable*}). Therefore, we re-assessed the accuracy of the approach we used in this different context. To do this, three raters manually validated a statistically significant sample of the transitions from our dataset.

To estimate the underlying probabilities that a file moves from one state to another, we measured the code readability of all the versions of source code files taken from the history of 25 software systems with ∼83k commits. We studied the evolution of readability at the commit level: this is the finest-grained analysis possibly achievable looking at the revision history of a software project.

Our results suggest that unreadable files are a minority and that most of them are unreadable since their introduction in the repositories. We observed a low readability deterioration: in all the projects analyzed, we found that unreadable files are more likely to become readable than the other way around.

We also manually analyzed the files for which the readability score varied the most throughout the history of the project, to understand (i) which types of changes (*i.e.,* perfective, corrective, or adaptive) affect readability the most, and (ii) why readability changes. We observed that the perfective and corrective changes we analyzed improved code readability. On the other hand, adaptive changes sometimes caused a significant readability reduction: most likely this happens when developers make big changes. Based on our results, we defined some guidelines that developers can adopt to keep the number of unreadable files low.

The remainder of this chapter is organized as follows. Section 7.2 presents the model we used to describe the readability evolution of a file. In Section 7.3 we report a preliminary empirical study in which we evaluate the performance of a state-of-the-art readability prediction model for readability evolution classification: we do this to understand to what extent existing models are reliable in this different context. In Section 7.4 we report the design and the results of our main empirical investigation, in which we analyze the readability evolution of software projects and we try to understand which changes mostly impact code readability. Finally, Section 7.5 discusses the threats to validity of the studies and Section 7.6 concludes this chapter.

7.2 Modeling Code Readability Evolution

Developers use Version Control Systems (VCSs) to track evolution of software projects. Different kinds of changes can be made to the source code: new features are introduced (*adaptive*), errors are fixed (*corrective*) and the whole code structure and quality is improved (*perfective*). Regardless of their type, changes may also directly or indirectly affect code readability. Having a model that allows to track the evolution of such a source code property may benefit practitioners in many ways: for example, it can help developers while performing code reviews, *i.e.,* a warning may be raised when code readability deteriorates, or it can allow project managers understanding how the code is evolving and when actions are needed.

Readability can be assessed at many granularity levels, ranging from small snippets to whole modules or systems; anyway choosing the right granularity level to model is not trivial. Having a fine-grained model (*e.g.,* tracking readability at method-level) would benefit developers when reviewing code and it would help them understanding how single parts are evolving; but methods often appear and disappear, they can be splitted and, therefore, keeping track of the changes would be hard. Furthermore, having a coarse-grained model (*e.g.,* tracking readability at module/system-level) would be mostly helpful for project managers, since it would give a generic idea on the health status of the project, and it would allow to have longer tracks, since modules/systems appear and disappear more rarely; on the contrary, this would provide small benefit to practitioners when developing or reviewing code. We chose to model readability at file-level. Files are the smallest units tracked by VCSs: it is easy to track their evolution, and they would be reasonably fine-grained to help developers as well.

Before choosing the granularity level, it is important to choose *how* to measure code readability to model code evolution. A readability score can be used: the available readability prediction tools [59, 199], by default, for each given artifact are able to output a continuous value ranging between 0 and 1. Such a score represents the *probability* inferred by the classifier that the specified file belongs to the class *readable*: as previously mentioned in Chapter 3, such approaches are based on *classification*, *i.e.,* they are designed to determine if a snippet is *readable* or *unreadable*. There is no empirical evidence that such scores reflect the source code readability level and, to the best of our knowledge, there is no continuous readability score for source code available in the literature. Having an automated estimation of code readability is essential for tracking code readability evolution since it would be impractical asking developers to manually evaluate the readability of all the versions of all the files of a software system. For this reason, we choose to use a discrete model and, specifically, we model the code readability evolution of a file using a state diagram.

Let us consider a project P and its revision history, $\{P_0, \ldots, P_l\}$. A source file f can be in four states in a given revision P_i:

1. *non-existing*, if the file does not exist in P_i;

2. *other-name*, if the file existed in the last revision, P_{i-1}, but with a different name: this helps to detect both renaming and move operations;

3. *readable*, if the file exists in P_i and it is readable;

4. *unreadable*, if the file exists in P_i and it is unreadable.

The initial state of a file can be either *non-existing* or *other-name*. When a file is created, there is a transition from *non-existing* to either *readable* or *unreadable*, depending on its readability. When a file f is renamed or moved to f_{new}, the initial state for f_{new} is *other-name* and the final state is *readable* or *unreadable*. It is worth remarking that VCSs such as git, on which our studies described in Sections 7.3 and 7.4 are based, do not explicitly keep track of the renaming/move operations. On the other hand, git is able to detect renaming and move operations when they occur: the heuristic used by git is based on textual similarity. Regardless of the actual name, if in P_{i-1} there are two files, `foo` and `bar`, and in P_i `foo` is renamed in `bar` and `bar` is removed, git detects the renaming/move from `foo` to `bar` instead of keeping the track from $\texttt{bar}_{i-1}$ to $\texttt{bar}_i$, which are different files. In general, the renaming operations occur when (i) a file is renamed/moved, or (ii) a folder which includes a file is renamed/moved. Even if git achieves good results in tracking file renaming operations, if two files `foo` and `bar` are similar enough and they are both renamed in the same commit (`foo` to `foo2` and `bar` to `bar2`), git could detect erroneous renaming operations, *e.g.,* from `foo` to `bar2` and from `bar` to `foo2`. For this reason, we use two different initial states (*i.e., non-existing* and *other-name*) to avoid mixing the two operations. We did not take into account file deletion operations (*i.e.,* from either *readable* or *unreadable* to *non-existing*): we assume that such an operation does not depend on the readability of a file. Instead, we assume that file deletions are rather mostly triggered by other needs, *e.g.,* a feature is no longer needed. Finally, every change which is not a *creation*, *renaming/move* or *deletion* operation, results in a transition from $\{readable/unreadable\}$ to $\{readable/unreadable\}$.

Given a revision P_i, we can safely assume that the state of a file in P_i only depends on the previous revision P_{i-1}. In other words, when developers work on a file, they reasonably react to the current state of the file and not to past states. For example, when a bug is fixed, this happens because the file contained a bug in P_{i-1}, regardless of the fact that the bug could be also in previous versions. Even if code from past revisions can be reused in some circumstances (*e.g.,* commits can be reverted), this always happens in reaction to specific properties of the working revision. This is true also for readability evolution: the fact that a file becomes readable or unreadable depends on the current readability of the file rather than on its past readability. Therefore, we can say that the readability evolution process is *memoryless* and it satisfies the Markov property. This allows us to define our readability evolution model as a Markov chain.

A Markov chain is a stochastic process in which the probability of transitioning from a state A to a state B does not depend on states attained in the past, but only on the last state. Given a Markov chain that can attain the states $\{S_1, \ldots, S_n\}$, for each couple of states S_i and S_j there is a probability $P(S_j|S_i)$ of transitioning

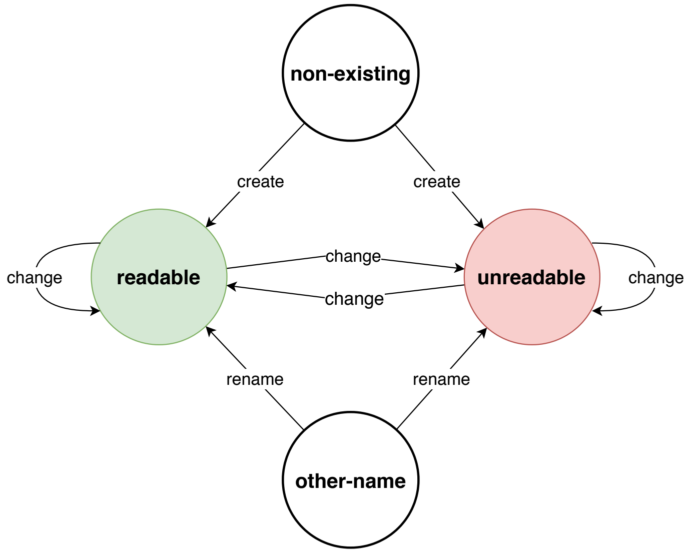


Figure 7.1: States of a source code file.

from S_i to S_j. Such probabilities are usually represented in a transition matrix, *i.e.,* a square matrix in which both rows and columns indicate the states and a given cell (i, j) contains the probability $P(S_j|S_i)$. Transitions not allowed have probability 0 in the transition matrix. The sum of each row of the matrix must be equal 1. We use a time-homogeneous Markov chain for our model: we assume that transition probabilities are constant in the time for a given project. This allows us to have a single transition probability for each pair of states, *i.e.,* S_i and S_j. While such an assumption may not always hold in practice (*e.g.,* the probability that a file is created *unreadable* may change with the evolution of a project), it helps us building a model that is easier to understand. Generic discrete-time Markov chains can be explored in future works in order to provide more fine-grained probabilities.

Figure 7.1 depicts the state diagram behind the Markov model we defined. The Markov chain we use to model the readability evolution process requires the estimation of the conditional probabilities associated with each transition (*i.e.,* the transition matrix). Defining the transition matrix would allow us to understand what is the probability that a file is created readable or unreadable, that a readable file becomes unreadable and vice versa and whether the fact that a file existing in the past affects the probability that it is readable. We describe the process we used to infer the probabilities of the readability evolution model of a given project in Section 7.4.

7.3 Study I: Validation of Readability Prediction in Software Evolution

The *goal* of our first study is to understand if readability prediction models, which were typically experimented in the context of single snippets of code, are suited for predicting readability evolution. As previously mentioned, the two problems are different: while state-of-the-art readability models are binary, *i.e.,* they classify a snippet as *readable* or *unreadable*, the problem we try to tackle is a 8-class classification problem, where each type of transition previously mentioned in Section 7.2 is a class. This preliminary study will allow us to better frame the main study, reported in Section 7.4.

Our first study is guided by the following research questions:

RQ_1 *Which readability values lead to classification errors?* While the readability prediction model we use is binary by definition, the tool returns the probability that the given snippet is readable (according to the underlying logistic model). There may be ranges of values for which the accuracy is not good enough. For example, 0.51 would indicate that, while the prediction is *readable*, there is still a 49% chance that it is *unreadable.* This research question aims at determining the range of readability values measured by the state-of-the-art readability classification approach in which the model wrongly classifies transitions. We later use the results of this analysis for filtering out the transactions on which the model is not accurate enough

Table 7.1: Projects considered in our study.

Project	**Repository URL**	**Commits**	**LOC**
Fullcontact4j	`https://github.com/fullcontact/fullcontact4j`	**234**	**6K**
Hibernate Metamodel Generator	`https://github.com/hibernate/hibernate-metamodelgen`	**173**	**10K**
NITHs	`https://github.com/niths/niths`	**1,396**	**24K**
Apache Qpid	`https://github.com/apache/qpid`	**3,367**	**25K**
JBoss Modules	`https://github.com/jboss-modules/jboss-modules`	**790**	**33K**
JBoss Tools JBPM	`https://github.com/jbosstools/jbosstools-jbpm`	**283**	**37K**
Nuxeo Runtime	`https://github.com/nuxeo-archives/nuxeo-runtime`	**1,174**	**55K**
Apache Incubator-Skywalking	`https://github.com/apache/incubator-skywalking`	**2,533**	**47K**
hlt-confdb	`https://github.com/cms-sw/hlt-confdb`	**1,040**	**72K**
ParSeq	`https://github.com/linkedin/parseq`	**454**	**75K**
Xnio	`https://github.com/xnio/xnio`	**1,096**	**77K**
OpenEngSB	`https://github.com/openengsb/openengsb`	**4,896**	**92K**
Apache Deltaspike	`https://github.com/apache/deltaspike`	**1,541**	**125K**
SIB-dataportal	`https://github.com/SIB-Colombia/sib-dataportal`	**104**	**137K**
Apache Falcon	`https://github.com/apache/falcon`	**1,755**	**154K**
IGV	`https://github.com/chenopodium/IGV`	**2,351**	**157K**
Undertow	`https://github.com/undertow-io/undertow`	**3,820**	**178K**
Apache Isis	`https://github.com/apache/isis`	**3,529**	**266K**
RxJava	`https://github.com/ReactiveX/RxJava`	**4,014**	**332K**
Apache Beam	`https://github.com/apache/beam`	**5,373**	**376K**
Apache Qpid-broker-j	`https://github.com/apache/qpid-broker-j`	**6,530**	**390K**
Apache Tomcat	`https://github.com/apache/tomcat`	**15,045**	**468K**
Apache Cxf	`https://github.com/apache/cxf`	**8,532**	**837K**
Apache Flink	`https://github.com/apache/flink`	**5,327**	**880K**
Apache Hadoop	`https://github.com/apache/hadoop`	**7,780**	**1.6M**
Total		**83K**	**6.5M**

from the dataset we introduce in this chapter: we do this to make the results of our main study are more reliable.

RQ_2 *Is the readability prediction model suited to assess the readability transitions?* This research question aims at measuring the accuracy of the readability prediction model we adopted to assess readability variation in the revision history of a software system.

7.3.1 Data Collection

The context of our study is constituted by archival data [196]. Specifically, we have studied the history of 25 Java open source projects. We report in Table 7.1 the projects we selected, along with the number of lines of code—in ascending order—in the last analyzed version. We chose projects with a reasonably big revision history (at least 100 commits) and big enough to encourage developers keeping the code readable (at least 5K LOC in their last revision). In total we considered the complete revision history of such projects until early 2018. Specifically, we considered all the commits from the master branch of each project [82]. In total, we focused our analysis on ∼83k commits.

Initially, for each project we extracted the history of each file f that ever appeared in its revision history. Given a project P and a file f that appeared in the revision history, we tracked its versions $\langle f_1, \ldots, f_n \rangle$. To achieve this goal, we analyzed the commit logs extracted from the git repositories of the projects. To do this, we only focused on Java source files (*i.e.,* files with extension `.java`). When a file was created, we started tracking this given file and measuring its readability: this resulted in the introduction of a new transition *non-existing* → {*readable*/*unreadable*}; when a file was modified, we measured its readability: in such cases, we added a new transition {*readable*/*unreadable*} → {*readable*/*unreadable*}; when a file was renamed, we introduced a new transition *other-name* → {*readable*/*unreadable*}.

We chose the tool by Scalabrino *et al.* [199] since it implements a comprehensive model for automatic code readability assessment, *i.e.,* the one which achieves the highest classification accuracy on all the datasets currently available (based on the comparison performed with all the other state-of-the-art tools reported by Scalabrino *et al.* [199]).

All the approaches available in the literature, including the one we used, were validated on small snippets (*e.g.,* methods) [58, 59, 181, 80, 200, 199]. In the current study we want to estimate the readability of classes instead. Considering the whole classes as snippets could mislead the classifier, since it is not trained on such samples. For example, one of the features used in the model measures the consistency between method-level comments and identifiers: while it would be possible to measure such a feature at class-level by merging all the comments for all the methods, there is no evidence that this feature is useful as well. To compute the readability of a given class C with n methods $C_1, \ldots, C_n$ we had several options for aggregating the readability computed at method-level by the tool. We used the arithmetic mean of $C_1, \ldots, C_n$ to estimate the readability of C. The main drawback of using mean is that it would not work properly in the cases in which there are many readable methods (*e.g.,* getters and setters) and a single unreadable method [228]. However, it is worth noting that any aggregation would have possibly distorted the readability predicted for the class and result in a classification error. We discuss in Section 7.5 other alternatives that we discarded. Since we aggregate the code readability measured at method-level, we exclude all the Java interfaces, which usually only define the method signatures.

We excluded from our study the interfaces and the enums, which usually do not provide the implementation of methods. We trained the classifier of Scalabrino *et al.* [199] with all the Java snippets and these are from the union of three readability datasets currently available [59, 80, 200], also performing features selection, as suggested in the original paper. The tool and the datasets are the original ones released by the authors, publicly available[1].

The tool we used to estimate the readability of a class returns a value between 0 and 1 for a given snippet. Such a number indicates the probability that the snippet is readable according to the logistic regression model: a readability of 1 means that the classifier is confident that the snippet is readable, while a readability of 0 means that the model is confident that the snippet is unreadable. In general, a value greater than 0.5 means that it is more likely that the snippet

[1] `https://dibt.unimol.it/report/readability/`

is readable rather than unreadable. Therefore, we use 0.5 as a natural threshold: we say that a file is *readable* if its readability is greater than or equals to 0.5 and *unreadable* otherwise.

7.3.2 Experimental Procedure

To answer RQ_1, we used the dataset provided by Pantiuchina *et al.* [170]. Such a dataset includes 1,282 commits in which the developers *explicitly* mentioned that they improved code readability. The dataset includes readability values measured before the commit (R_{before}) and after (R_{after}), besides other metrics. As a first step, we associated to each commit the corresponding transition (*i.e., {readable/unreadable}* → *{readable/unreadable}*) based on the R_{before} and R_{after}, using the same procedure we used to build our dataset. Then, we extracted from such a dataset all the readability transitions classified by the tool as *readable* → *unreadable* (*i.e.,* so that $R_{before} \geq 0.5$ and $R_{after} < 0.5$). These are the only cases for which we are reasonably sure that the tool made a classification mistake. Indeed, if the predicted transition is *unreadable* → *readable*, this agrees with what developers claimed; on the other hand, if the predicted transition is *readable* → *readable* it can still be true that there was an improvement in readability; even if the predicted transition is *unreadable* → *unreadable*, again, there might have been an improvement to some methods, but not big enough to make the file become totally *readable*.

Given the subset of transitions on which the tool strongly disagrees with the developers' claims, we manually analyzed such transitions and excluded the ones on which the tool clearly did not make a mistake (*i.e.,* the developer said that the readability increased but it actually decreased). To do this, two of the authors of the study (with 7 and 10 years of Java programming experience) independently analyzed all the selected transitions and they openly discussed the ones on which at least one of them disagreed with the commit message. We discuss such cases in the results and we explicitly mention the reason why we decided that, for such cases, the developers were wrong. Finally, we considered the range $[min(R_{after}), max(R_{before})]$ as the range in which the tool most likely makes classification mistakes: excluding transitions which involve readability values within this range would allow us to have no explicit classification mistakes on the dataset by Pantiuchina *et al.* [170], for which we have an oracle provided by the developers themselves. We use $min(R_{after})$ as lower-bound because the tool classified the class as *unreadable* after the commit, *i.e.,* R_{after} will be lower than 0.5; similarly, we use $max(R_{before})$ as upper-bound since the tool classified the class as *readable* before the commit, *i.e.,* R_{before} will be higher than 0.5. We excluded all the transitions in our dataset that had a readability value in such a range to minimize the classification error in all the other research questions.

To answer RQ_2, we considered a significant random stratified sample of our dataset. Such a sample contained 271 transitions out of the total 346,337 (90% confidence level, 5% confidence interval). The strata of the sample were all the eight possible transitions types (*i.e., created* → *readable*, *created* → *unreadable*, *other-name* → *readable*, *other-name* → *unreadable*, *readable* → *unreadable*, *unreadable* → *readable*, *unreadable* → *unreadable*, *readable* → *readable*).

Three of the authors of the study, (with 5, 7, and 10 years of Java programming experience), independently reported their agreement with both the binary readability values computed by the tool for each transition (*i.e.,* R_{before} and R_{after}). To do this, we used a 5-point Likert scale from -2 to +2, where "-2" means "I totally disagree with the tool" (*e.g.,* if the tool says that the commit is *readable*, the evaluator thinks that it is *unreadable* without any doubt) and "+2" means "I totally agree with the tool" (*e.g.,* if the tool says that the commit is *readable*, the evaluator thinks that it is *readable* without any doubt).

In a first phase, at least two authors of the study evaluated each transitions from the sample. Then, all the evaluators discussed the cases in which there was a disagreement between the two evaluators involved in the first phase and after that, they resolved the disagreements in an unanimous manner. In Section 7.3.3 we report the details about the scores given by the annotators.

After performing a manual classification, we determined that transitions occurred in the following way: we kept the binary values of R_{before} and R_{after} when the authors agreed with them (evaluation greater than 0) and we swapped them when the evaluators disagreed (evaluation lower than 0). For example, if the tool classified a given transition as *readable* → *unreadable* and the manual evaluation was (-2, +2), we inferred that the actual transition occurred was *unreadable* → *unreadable* (*i.e.,* we swapped the first value).

Therefore, at the end of the manual evaluation, we had both a transition automatically predicted by the tool and an oracle transition. We report *precision* and *recall* on this sample of transitions for each class we took into account (*i.e.*, each transition type), using the following formulas:

$$
\begin{aligned}
precision_t &= \frac{TP_t}{TP_t + FP_t},\\
recall_t &= \frac{TP_t}{TP_t + FN_t},\\
F_t &= 2 * \frac{precision_t * recall_t}{precision_t + recall_t}\ ,
\end{aligned}
$$

where:

- TP_t (or *true positive* for transition t) indicates the number of cases for which the tool classifies a transition as t and our evaluation confirms that;
- FP_t (or *false positive* for transiton t) indicates the number of cases for which the tool classifies a transition as t and our evaluation does not confirm that;
- FN_t (or *false negative* for transiton t) indicates the number of cases for which the tool classifies a transition as different from t and our evaluation does not confirm that;
- TN_t (or *true negative* for transition t) indicates the number of cases for which the tool classifies a transition as different from t and our evaluation confirms that.

We report in Table 7.2 a summary of the dataset we used to answer each research question in this first study. We provide a replication package [176] with (i) the dataset we built and (ii) the data used to answer RQ_1 and RQ_2.

7.3.3 Empirical Study Results

We report in this section the results of our Study I for each research question.

Table 7.2: Datasets used in the first study.

Research Question	Dataset	No. of transitions
RQ_1	Pantiuchina *et al.* [170]	1,282
RQ_2	Significant subset of our dataset	271

RQ_1: Which Readability Values Lead to Classification Errors?

We found 20 cases in which the readability model we used classified a transition from the dataset by Pantiuchina *et al.* [170] as *readable* → *unreadable* (*i.e.,* $R_{before} \geq 0.5$ and $R_{after} < 0.5$). After manually analyzing such cases, we excluded three of them: two file modifications[2] were excluded because we could not find the related commit in the repository: probably, such a commit was deleted by the project contributors. We excluded another commit[3]: the modification consisted exclusively in the deletion of Javadoc comments. It is not clear why removing documentation, specifically in that context, should have resulted in higher readability. For this reason, we excluded such a change.

After filtering out such three data-points, we found that the interval in which the tool makes all the mistakes is $[min(R_{after}) = 0.416, max(R_{before}) = 0.600]$. From a initial dataset of 457,651, we excluded 111,314 transitions with readability values in this range from our dataset, obtaining a new dataset with 346,337 transitions. We exclude a conspicuous portion of our dataset (∼24% of the transitions) because of this filtering. However, we think that, in this context, it is more important having a reasonably reliable measure rather than a large number of data-points, also given the fact that we still have many data-points to analyze. Moreover, besides allowing us to make no mistakes on the dataset by Pantiuchina *et al.* [170], the range we filter out is intuitively reasonable: we exclude transitions on which, according to the model, there is more than ∼40% chance of error. For example, if the predicted readability is 0.58, it means that the file is readable, but there is a 42% chance that it is unreadable (*i.e.,* the prediction is wrong): we exclude such cases. It is worth highlighting that this does not mean that outside

[2] Commit `c69c7b` in the project android_packages_apps_Settings.

[3] Book-App-Java-Servlet-Ejb-Jpa-Jpql, commit 36861: `https://git.io/fjLfp`

Table 7.3: Confusion matrix on the whole evaluated sample

		Our evaluation							
		R → ***R***	***R*** → ***U***	***U*** → ***R***	***U*** → ***U***	***NE*** → ***R***	***NE*** → ***U***	***ON*** → ***R***	***ON*** → ***U***
Tool	***R*** → ***R***	29	-	-	5	-	-	-	-
	R → ***U***	10	22	2	-	-	-	-	-
	U → ***R***	6	-	19	8	-	-	-	-
	U → ***U***	3	3	-	28	-	-	-	-
	NE → ***R***	-	-	-	-	30	4	-	-
	NE → ***U***	-	-	-	-	14	20	-	-
	ON → ***R***	-	-	-	-	-	-	34	-
	ON → ***U***	-	-	-	-	-	-	16	18

Table 7.4: Performance of the tool in transition classification.

Transition	**Precision**	**Recall**	**F-measure**
non-existing → readable	88.2%	68.2%	76.9%
non-existing → unreadable	58.8%	83.3%	69.0%
other-name → readable	100.0%	68.0%	81.0%
other-name → unreadable	52.9%	100.0%	69.2%
readable → readable	85.3%	60.4%	70.7%
readable → unreadable	64.7%	88.0%	74.6%
unreadable → readable	57.6%	90.5%	70.4%
unreadable → unreadable	82.4%	68.3%	74.7%

such a range the tool is necessarily accurate: we only identified the range in which the readability prediction was most unreliable.

Summary of RQ_1**.** The tool by Scalabrino *et al.* [199] is not reliable when the output is in the range [0.416, 0.600].

RQ_2: Is the Readability Prediction Model Suited to Assess the Readability Transitions?

The raters disagreed in the evaluation they performed in 46 cases out of the 407 cases analyzed (11.3% of the cases), including the evaluation of the readability both before and after the commit: as previously mentioned, such cases were discussed by all the evaluators and consensus was reached on all of them. As for single snapshots, the raters agreed with the tool in 82% of the cases. This result is in line with the accuracy reported in the original study (*i.e.,* ∼84%).

As for transitions, we report in Table 7.3 the *confusion matrix* for the 8-class categorization problem. Also, in Table 7.4 we report precision, recall, and F-measure obtained on the samples we manually evaluated. We found that the tool by Scalabrino *et al.* [199] has a high precision in classifying some transitions, mostly the ones in which it is involved the state *readable* (*e.g.,* the transition *non-existing* → *readable* has the 88.2% of precision with the 68.2% of recall), while it has a high recall for other transitions, *e.g., unreadable* → *readable*, with 90.5% of recall and 57.6% of precision.

In general, the results show that the tool is less accurate when it is used to classify transitions compared to when it is used to classify single versions of a file. This may be due to the fact that classification problem has two possible cases: a case happens when the file already exists and a case when the file is created or renamed. Indeed, if the file already exists, we have four possible classes (*i.e., readable* → *readable*, *readable* → *unreadable*, *unreadable* → *readable*, *unreadable* → *unreadable*). Instead, if the file is new or is renamed, the classification problem involves two classes (*i.e., non-existing (or other-name)* → *readable* and *non-existing (or other-name)* → *unreadable*). The tool needs to correctly classify two versions of a file to rightly classify a transition, which is harder than correctly classifying just a single snapshot. It is worth noting that it is not trivial correctly classifying even transitions *readable* → *readable* or *unreadable* → *unreadable*: it is necessary to correctly predict two different snapshots of the same snippet, and a difference even in a single feature used by the underlying model (*e.g.,* line length) could possibly confuse the model. Looking at the confusion matrix, it can be noticed that the tool often confuses *readable* → *readable* transitions with *readable* → *unreadable* ones. For example, this happens for a change to the class `SpdySynStreamStreamSourceChannel` of Undertow[4]: in this case, some methods were removed and, while the remaining ones are clearly readable, the tool wrongly classified the constructor, which has a very long line for the signature (167 characters). Probably, the same mistake was done in the previous version, but the other methods avoided to wrongly classify the whole class.

[4] Undertow, commit `f8fcc`: `https://git.io/Jf1pt`

Summary of RQ_2. There is a loss of accuracy when using readability prediction tools for transitions instead of single versions.

7.4 Study II: Readability Evolution

The *goal* of our empirical study is to understand how readability evolves in software development, *i.e.,* how frequently code readability changes and why it changes. The *perspective* is that of a researcher who wants to understand how readability is managed in practice and that of a software quality consultant who wants to recommend how to avoid readability deterioration.

Our empirical study is guided by the following research questions:

RQ_3 *How often does readability change?* This research question aims at understanding how frequently readable code becomes unreadable and vice versa. Considering Q_3 in Section 6.2, with this research question we also want to verify if there is a match between what people *say* with what people *do* [82, 83].

RQ_4 *How and why does code readability change?* With this research question we take a closer look at the source code modifications leading to readability changes in order to understand (i) which kind of changes make code readability evolve and (ii) why code readability changes.

7.4.1 Experimental Procedure

To answer both our research questions, we used the dataset we collected (described in Section 7.3.1). We report in Table 7.5 a summary of the datasets we have used to answer each research question.

Table 7.5: Dataset used in the second study.

Research Question	Dataset	No. of transitions
RQ_3	Our new dataset (with bootstrapping)	346,337
RQ_4	Subset of our dataset	57

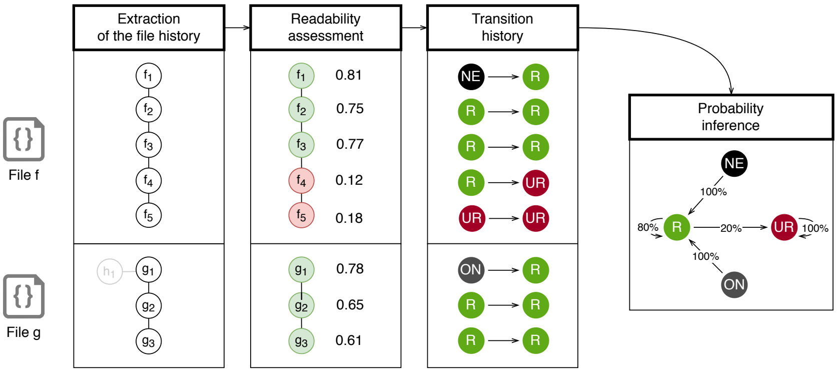


Figure 7.2: The process we used to compute the transition probability of a project.

To answer RQ_3, considering the history of each file f we associated to each revision a transition in the readability evolution model described in Section 7.2. We perform such an analysis at commit-level, *i.e.,* at the finest-grained level achievable when looking at the revision history of a project. Other choices could have been made, such as considering a more coarse-grained level (*e.g.,* release-level). We think that the finest-grained analysis is more useful for developers who want to continuously check if readability is deteriorating: for example, such a model could be used in Continuous Integration pipelines to allow developers finding issues and fixing them as soon as possible.

File modifications do not necessarily change the readability, *i.e.,* the model contains self-loops from *readable* to *readable*, and from *unreadable* to *unreadable*. Figure 7.2 summarizes the procedure we used to compute the transition probability of a project.

We counted the frequencies of all the transitions we observed in each project and used them to compute the probabilities of transition in code readability evolution model. We estimated the probability $P(S_j|S_i)$ as:

$$P(S_j|S_i) = \frac{freq(S_i \to S_j)}{\sum_{S_k} freq(S_i \to S_k)}$$

We report the probabilities we inferred from our data for all the projects taken individually. We also report the percentage of files that were always readable (f_i readable $\forall f_i$), always unreadable (f_i unreadable $\forall f_i$), and that changed readability ($\exists f_i$ readable and $\exists f_j$ unreadable). To account for the possible errors made by the tool, we used bootstrapping [78] to compute the confidence intervals of each probability.

Specifically, we used the m-out-of-n bootstrap [69], where n is the original sample size and m is lower than n. We do not use the *original bootstrapping*, because our samples contain many data points (346,337), and we set $m = 0.8n$ [69]. Therefore, we extracted 10,000 subsamples with repetition of m transitions file from our dataset for each single project and, for each of them, we estimated $P(S_j|S_i)$. Given the resulting distribution, we report the 90% confidence interval, *i.e.,* the 5% and 95% quantiles.

Furthermore, we try to understand if there is a correlation between the percentage of files that are created unreadable and remain unreadable with the number of commits of the projects, the number of files, and the number of contributors at the last commit we analyzed. To do this, we report the Kendall's τ, along with the p-values, correct for multiple comparisons using the method of Benjamini and Hochberg [50]. The p-values for correlations represent evidence against the null-hypothesis that the correlation between the ranks of the variables we study equals 0 [204]. Therefore, we use p-values only as a sanity check: significant correlation may still be very low and practically nonexistent.

Finally, we specifically focus on files that experienced a change in terms of code readability and we try to verify if it is possible to characterize such changes in terms of number of files modified in the commit and number of changed lines. As a first step, we extract the number of modified files and changed lines (added or removed) from each commit of the revision history of each project. We did this considering only files that had at least a readability increase or decrease in their history (*i.e., readable* → *unreadable* or *unreadable* → *readable*). Then, we divided the data we gathered in three groups:

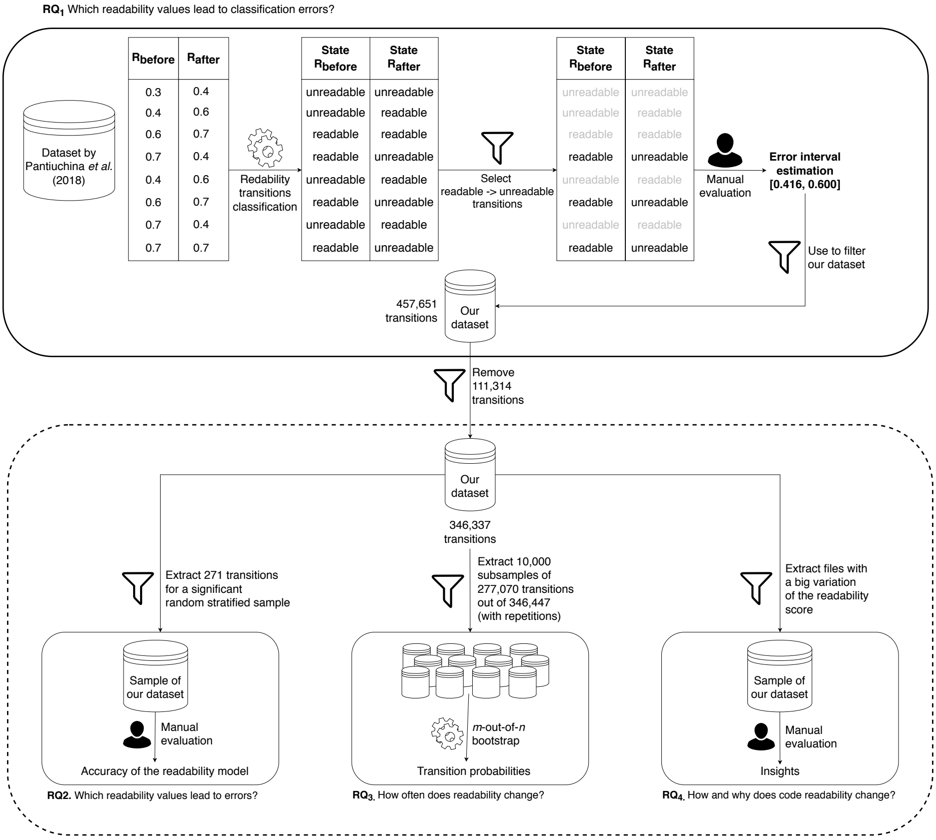
- R^+: transitions representing an increase of code readability (*i.e.,* transitions *readable* → *unreadable*);

- R^-: transitions representing a decrease of code readability (*i.e.,* transitions *unreadable* → *readable*);
- R^0: transitions that did not result in a readability change (*i.e.,* transitions *readable* → *readable* and *unreadable* → *unreadable*).

Then, we formulate three hypotheses for each property we compare (*i.e.,* number of modified files, number of changed lines, number of added lines, number of removed lines): (i) *there is a difference between commits that increase readability (R^+) and commits that decrease readability (R^-)*; (ii) *there is a difference between commits that increase readability (R^+) and commits that do not change readability (R^0)*; (iii) *there is a difference between commits that decrease readability (R^-) and commits that do not change readability (R^0).* We use the the Wilcoxon rank-sum test [236] to check such hypotheses: specifically, the null hypotheses are that there is no significant difference between such pairs of groups. We performed the tests for each project separately and also for all the whole dataset. We do not include in the comparison groups containing the states *created* and *other-name* since we are interested in characterizing changes in the evolution rather than at the introduction of a file. We reject a null hypothesis if the p-value is lower than 0.05. We adjust the p-values obtained for the group of hypotheses related to each metric using the Benjamini and Hochberg [50] method. We also compute the effect size to quantify the magnitude of the significant differences we find. We use Cliff's delta [71] since it is non-parametric. Cliff's d lays in the interval [-1, 1]: the effect size is **negligible** for $|d| < 0.148$, **small** for $0.148 \leq |d| < 0.33$, **medium** for $0.33 \leq |d| < 0.474$, and **large** for $|d| \geq 0.474$. If $d > 0$, it means that the first group is larger than the second, while the opposite happens otherwise.

To answer RQ_4, we selected the files that, during the history of the projects, had a big variation in code readability in terms of the continuous readability score, *i.e.,* files with minimum readability lower than or equal to 0.25 and maximum readability higher than or equal to 0.75. Two of the authors manually analyzed the history of these files. For each change we annotated:

- the type of change that the developer did to the specific file among *adaptive* (new feature implementation), *corrective* (bug fixing), and *perfective* (refactoring and code cleaning);


RQ$_1$ Which readability values lead to classification errors?
Dataset by Pantiuchina et al. (2018)
Redability transitions classification
Select readable -> unreadable transitions
Manual evaluation
Error interval estimation [0.416, 0.600]
Use to filter our dataset
457,651 transitions
Our dataset
Remove 111,314 transitions
Our dataset
346,337 transitions
Extract 271 transitions for a significant random stratified sample
Extract 10,000 subsamples of 277,070 transitions out of 346,447 (with repetitions)
Extract files with a big variation of the readability score
Sample of our dataset
Manual evaluation
Accuracy of the readability model
RQ2. Which readability values lead to errors?
m-out-of-n bootstrap
Transition probabilities
RQ$_3$. How often does readability change?
Sample of our dataset
Manual evaluation
Insights
RQ$_4$. How and why does code readability change?

R_{before}	R_{after}
0.3	0.4
0.4	0.6
0.6	0.7
0.7	0.4
0.4	0.6
0.6	0.7
0.7	0.4
0.7	0.7

State R_{before}	State R_{after}
unreadable	unreadable
unreadable	readable
readable	readable
readable	unreadable
unreadable	readable
readable	unreadable
unreadable	readable
readable	unreadable

State R_{before}	State R_{after}
unreadable	unreadable
unreadable	readable
readable	readable
readable	unreadable
unreadable	readable
readable	unreadable
unreadable	readable
readable	unreadable

Figure 7.3: Process of Study I and Study II.

- why readability changed (*e.g.,* comments added or long lines removed), focusing on three aspects: *visual*, *structural*, and *textual* [199].

In case of disagreement, two authors of the study performed an open discussion to resolve conflicting cases. We used the results of the analysis to formulate suggestions, available in Section 7.4.2, on how to avoid readability deterioration and annotations in order to improve source code readability.

In Figure 7.3 we summarize the process used for our two studies. Our replication package [176] contains the datasets we used to answer RQ_3 and RQ_4 to foster the replicability of this study.

7.4.2 Empirical Study Results

We report in this section the results of our Study II for each research question.

RQ_3: How often Does Readability Change?

Tables 7.6 and 7.7 show the probabilities estimated for the 25 projects we considered. We report in the table the mean probability estimation over the 10,000 bootstrap subsamples, along with the 90% confidence intervals below each estimation, in the form [5% quantile, 95% quantile]. The vast majority of the files are created *readable*. In eighteen projects out of twenty-five, the probability of creating an unreadable file is lower than 10%. However, there are exceptions: in Apache Incubator-Skywalking and Apache Beam, about a quarter of the files are created *unreadable*, while in ParSeq this percentage is even higher (more than a 40%). We discuss some examples for such a project later.

In general, we did not observe significant differences for file renaming as compared to file creation in terms of the resulting readability; for some projects such as Apache Qpid and Apache Flink, however, the probability of transitioning from *other-name* to *unreadable* is higher as compared to the probability of transitioning from *non-existing* to *unreadable*. For such projects, class rename/move refactoring operations are either more likely to be performed on unreadable files or they are performed while also changing other aspects of the source code, which make such files unreadable.

File modifications rarely result in a change in code readability. Usually, *readable* files remain *readable*, while *unreadable* files remain *unreadable*. We observed that, generally, it is more likely that *unreadable* files become readable than the opposite. In some projects, such as Apache Tomcat, IGV, Xnio, and hlt-confdb, readability improvement seems a priority: the probability that unreadable files become readable is higher than 10%. It is worth noting that such projects also achieve low probabilities of introducing unreadable files. Nevertheless, it is still likely that such a phenomenon is unconscious in such projects, *i.e.,* the developers do not make an effort to improve readability, but it is a side effect of their regular work, as reported in previous studies [44, 65, 224, 208, 141, 241]. We provide more details about this aspect in our qualitative analysis.

Table 7.6: Mean readability evolution probabilities and 90% confidence intervals of the bootstrap subsamples (file introduction).

Project	***non-existing→***		***other-name→***	
	readable	***unreadable***	***readable***	***unreadable***
Apache Beam	78.4% [77.3%, 79.6%]	21.6% [20.4%, 22.7%]	73.7% [71.0%, 76.4%]	26.3% [23.6%, 29.0%]
Apache Cxf	90.7% [90.1%, 91.3%]	9.3% [8.6%, 10.0%]	97.8% [93.0%, 100%]	4.1% [2.3%, 8.1%]
Apache Deltaspike	93.4% [92.2%, 94.5%]	6.6% [5.5%, 7.8%]	100.0% [100.0%, 100.0%]	0.0% [0.0%, 0.0%]
Apache Falcon	87.4% [85.9%, 89.0%]	12.5% [11.0%, 14.1%]	91.8% [86.7%, 96.3%]	8.2% [3.7%, 13.3%]
Apache Flink	81.3% [80.5%, 82.1%]	18.7% [17.9%, 19.5%]	64.9% [58.8%, 70.8%]	35.1% [29.2%, 41.2%]
Apache Hadoop	91.4% [90.8%, 92.0%]	8.6% [8.0%, 9.2%]	96.6% [94.4%, 98.6%]	3.4% [1.4%, 5.6%]
Apache I.-Skywalker	75.8% [75.0%, 76.5%]	24.2% [23.4%, 25.0%]	75.6% [73.6%, 77.5%]	24.4% [22.5%, 26.4%]
Apache Isis	93.0% [92.5%, 93.5%]	7.0% [6.4%, 7.5%]	90.4% [87.8%, 92.7%]	9.6% [7.2%, 12.1%]
Apache Qpid-b.	91.1% [90.5%, 91.7%]	8.9% [8.2%, 9.5%]	100.0% [100.0%, 100.0%]	0.0% [0.0%, 0.0%]
Apache Qpid	91.6% [90.8%, 92.4%]	8.4% [7.6%, 9.2%]	79.8% [61.5%, 94.7%]	21.0% [6.7%, 38.5%]
Apache Tomcat	95.7% [95.0%, 96.4%]	4.3% [3.6%, 4.9%]	100.0% [100.0%, 100.0%]	0.0% [0.0%, 0.0%]
Fullcontact4j	95.4% [93.2%, 97.4%]	4.6% [2.6%, 6.8%]	94.6% [87.5%, 100.0%]	6.8% [2.8%, 14.3%]
Hibernate Metamodel G.	97.6% [95.8%, 99.1%]	2.4% [0.9%, 4.2%]	//	//
hlt-confdb	91.4% [88.1%, 94.4%]	8.6% [5.5%, 11.9%]	//	//
IGV	93.5% [92.0%, 94.9%]	6.5% [5.1%, 8.0%]	100.00% [100.00%, 100.00%]	0.0% [0.0%, 0.0%]
JBoss Modules	91.9% [89.2%, 94.5%]	8.1% [5.5%, 10.8%]	100.00% [100.00%, 100.00%]	0.0% [0.0%, 0.0%]
JBoss Tools JBPM	95.2% [93.5%, 96.8%]	4.8% [3.1%, 6.5%]	//	//
NITHs	93.8% [92.0%, 95.5%]	6.2% [4.5%, 8.0%]	89.2% [82.1%, 95.4%]	10.8% [4.7%, 17.8%]
Nuxeo Runtime	93.4% [92.0%, 94.7%]	0.6% [0.5%, 0.8%]	93.6% [89.5%, 97.2%]	6.4% [2.8%, 10.5%]
OpenEngSB	89.8% [89.0%, 90.6%]	10.1% [9.4%, 10.9%]	92.1% [90.9%, 93.3%]	7.9% [6.7%, 9.1%]
ParSeq	58.7% [55.7%, 61.7%]	41.3% [38.3%, 44.3%]	81.3% [61.5%, 100.0%]	20.5% [7.1%, 40.0%]
RxJava	91.7% [91.0%, 92.5%]	8.3% [7.5%, 9.0%]	87.5% [84.5%, 90.3%]	12.5% [7.1%, 15.5%]
SIB dataportal	95.7% [94.4%, 97.0%]	4.3% [3.1%, 5.5%]	//	//
Undertow	78.9% [77.1%, 80.1%]	21.1% [19.3%, 22.8%]	97.8% [93.2%, 100.0%]	3.9% [2.2%, 7.9%]
Xnio	90.4% [88.0%, 92.7%]	9.6% [7.2%, 12.0%]	//	//

Table 7.7: Mean readability evolution probabilities and 90% confidence intervals of the bootstrap subsamples (file evolution).

Project	***readable→***		***unreadable→***	
	readable	***unreadable***	***readable***	***unreadable***
Apache Beam	99.8% [99.8%, 99.9%]	0.2% [0.1%, 0.2%]	0.6% [0.4%, 0.7%]	99.4% [99.3%, 99.6%]
Apache Cxf	99.8% [99.8%, 99.9%]	0.2% [0.1%, 0.2%]	1.9% [1.4%, 2.4%]	98.1% [97.6%, 98.6%]
Apache Deltaspike	99.8% [99.6%, 99.9%]	0.2% [0.1%, 0.4%]	4.9% [2.1%, 7.9%]	95.1% [92.1%, 97.9%]
Apache Falcon	99.6% [99.4%, 99.8%]	0.4% [0.2%, 0.6%]	0.9% [0.4%, 1.4%]	99.1% [98.6%, 99.6%]
Apache Flink	99.6% [99.6%, 99.7%]	0.4% [0.2%, 0.4%]	2.1% [1.6%, 1.4%]	97.9% [97.4%, 98.3%]
Apache Hadoop	99.9% [99.9%, 99.9%]	0.1% [0.0%, 0.1%]	0.9% [0.5%, 1.3%]	99.1% [98.7%, 99.4%]
Apache I.-Skywalker	99.4% [99.2%, 99.5%]	0.6% [0.4%, 0.7%]	0.7% [0.4%, 0.9%]	99.3% [99.1%, 99.5%]
Apache Isis	99.8% [99.8%, 99.9%]	0.1% [0.0%, 0.2%]	3.7% [2.6%, 4.8%]	96.3% [95.2%, 97.4%]
Apache Qpid-b.	99.9% [99.8%, 99.9%]	0.1% [0.0%, 0.2%]	1.9% [1.4%, 2.5%]	98.1% [97.5%, 98.6%]
Apache Qpid	99.9% [99.8%, 99.9%]	0.1% [0.0%, 0.1%]	2.5% [1.6%, 3.4%]	97.5% [96.6%, 98.4%]
Apache Tomcat	99.9% [99.9%, 99.9%]	0.1% [0.0%, 0.1%]	5.6% [3.8%, 7.5%]	94.4% [92.5%, 96.2%]
Fullcontact4j	99.5% [98.8%, 100.0%]	0.5% [0.2%, 1.2%]	0.0% [0.0%, 0.0%]	100.0% [100.0%, 100.0%]
Hibernate Metamodel G.	100.0% [100.0%, 100.0%]	0.0% [0.0%, 0.0%]	0.0% [0.0%, 0.0%]	100.0% [100.0%, 100.0%]
hlt-confdb	99.9% [99.8%, 100.0%]	0.1% [0.0%, 0.2%]	6.3% [2.3%, 11.2%]	93.8% [88.8%, 97.8%]
IGV	99.7% [99.5%, 99.8%]	0.3% [0.1%, 0.4%]	5.1% [2.5%, 7.9%]	94.9% [92.1%, 97.4%]
JBoss Modules	99.9% [99.8%, 100.0%]	0.1% [0.0%, 0.2%]	2.8% [1.2%, 5.7%]	97.8% [94.5%, 100.0%]
JBoss Tools JBPM	97.9% [96.3%, 99.2%]	0.2% [0.1%, 0.4%]	0.0% [0.0%, 0.0%]	100.0% [100.0%, 100.0%]
NITHs	99.9% [99.8%, 100.0%]	0.1% [0.0%, 0.2%]	3.0% [0.9%, 5.7%]	97.0% [94.3%, 99.2%]
Nuxeo Runtime	>99.9% [99.9%, 100.0%]	0.1% [0.0%, 0.1%]	2.9% [1.0%, 4.5%]	97.1% [95.6%, 100.0%]
OpenEngSB	99.8% [99.7%, >99.8%]	0.2% [0.1%, 0.3%]	1.6% [1.1%, 2.1%]	98.4% [97.9%, 98.9%]
ParSeq	99.1% [98.4%, 99.6%]	0.9% [0.4%, 1.6%]	0.8% [0.5%, 1.6%]	99.6% [97.9%, 98.9%]
RxJava	100.0% [99.9%, 100.0%]	<0.1% [0.0%, 0.1%]	2.4% [1.4%, 3.4%]	97.6% [96.6%, 98.6%]
SIB	99.7% [99.2%, 100.0%]	0.05% [0.03%, 1.1%]	0.0% [0.0%, 0.0%]	100.0% [100.0%, 100.0%]
Undertow	99.6% [99.5%, 99.8%]	0.4% [0.2%, 0.5%]	0.8% [0.4%, 1.2%]	99.1% [98.8%, 99.5%]
Xnio	99.7% [99.5%, 99.9%]	0.3% [0.1%, 0.5%]	5.1% [1.9%, 8.7%]	94.9% [91.3%, 98.1%]

The probabilities of readability changes we reported regard a single commit. It could be argued that readability changes are unlikely simply because most of the changes are small and thus they may affect readability only in the long run. To look more in depth into this, we also computed the percentage of files that (i) are created readable and remain readable, (ii) are created unreadable and remain unreadable, (iii) change readability at least once during the evolution. Table 7.8 shows the aforementioned results: only a minority (usually less than 5%) of the files change its readability during the history of a project, while 88.9% of the files remain readable and 9.9% of them are always unreadable, on average.

We found that the number of commits is significantly correlated with the number of unreadable files (Kendall τ **0.31**, corrected p-value **0.049**): this suggests that the longer the history of the project is, the higher the risk of introducing unreadable code in the project. Anyway, such a correlation is low, in absolute terms. We get a stronger correlation with the number of contributors: in this case, the Kendall τ is **0.43** and the corrected p-value is **0.009**: this suggests that the larger the development team, the higher the number of unreadable files. Finally, we found a weak and non-significant correlation with the number of files ($\tau = 0.21$, corrected p-value $= 0.16$): this suggests that the size of the project is not one of the most important factors that affects the percentage of unreadable files in a project. It is worth remarking that, since correlations do not imply causation, the real existence of the relationships we found using such a simple analysis should be properly verified with more in-depth analyses in future work.

Table 7.9 shows the results of our analysis regarding the characteristics of readability evolution transitions for all the data-points, while we report in Tables 7.10 and 7.11 the results at project-level. In Figure 7.4 we report the boxplots of files changed, lines changed, lines added and lines removed for each group, adjusted for skewed distributions [113].

It is possible to notice that there is a low number of significant differences in terms of number of changed files among the groups when taking into account single projects; on the other hand, there are significant differences when taking into account all the data-points. Such differences, however, are only negligible in terms of effect size. This shows that the number of files modified in a commit are not related to the presence of changes in code readability. Instead, it can

Table 7.8: Files always readable, always unreadable or both readable and unreadable in the revision history of the projects.

Project	**Readable**	**Unreadable**	**Both**
Apache Beam	79.0%	19.5%	1.5%
Apache Cxf	90.3%	8.6%	1.0%
Apache Deltaspike	93.2%	5.8%	0.9%
Apache Falcon	86.6%	12.3%	1.2%
Apache Flink	80.8%	17.7%	1.5%
Apache Hadoop	91.3%	8.2%	0.6%
Apache Incubator-S.	77.2%	21.4%	1.4%
Apache Isis	92.6%	6.7%	0.7%
Apache Qpid-broker-j	90.3%	8.8%	0.9%
Apache Qpid	91.4%	7.8%	0.8%
Apache Tomcat	95.1%	3.5%	1.4%
Fullcontact4j	93.0%	5.7%	1.3%
Hibernate Metamodel Gen.	97.6%	2.4%	0.0%
hlt-confdb	90.5%	6.4%	3.0%
IGV	92.1%	5.5%	2.4%
JBoss Modules	91.7%	6.8%	1.1%
JBoss Tools JBPM	94.2%	4.6%	1.2%
NITHs	93.0%	5.6%	1.4%
Nuxeo Runtime	94.4%	5.1%	0.5%
OpenEngSB	89.2%	9.4%	1.4%
ParSeq	64.7%	33.9%	1.4%
RxJava	92.3%	7.1%	0.6%
SIB-dataportal	95.9%	4.0%	0.1%
Undertow	77.1%	20.7%	2.2%
Xnio	89.2%	9.1%	1.7%

Table 7.9: Comparison of characteristics of the commits (number of files and lines changed) among different transaction types.

		Files changed				**Lines changed**	
Comparison		***p*-value**	**Cliff's *d***	**Comparison**		***p*-value**	**Cliff's *d***
R^+	R^-	0.308	//	R^+	R^-	0.153	//
R^+	R^0	0.060	//	R^+	R^0	**< 0.001**	**0.513 (large)**
R^-	R^0	**< 0.001**	-0.099 (negligible)	R^-	R^0	**< 0.001**	**0.527 (large)**
		Lines added				**Lines removed**	
Comparison		***p*-value**	**Cliff's *d***	**Comparison**		***p*-value**	**Cliff's *d***
R^+	R^-	**0.017**	-0.095 (negligible)	R^+	R^-	0.843	//
R^+	R^0	**< 0.001**	**0.385 (medium)**	R^+	R^0	**< 0.001**	**0.375 (medium)**
R^-	R^0	**< 0.001**	**0.478 (large)**	R^-	R^0	**< 0.001**	**0.319 (small)**

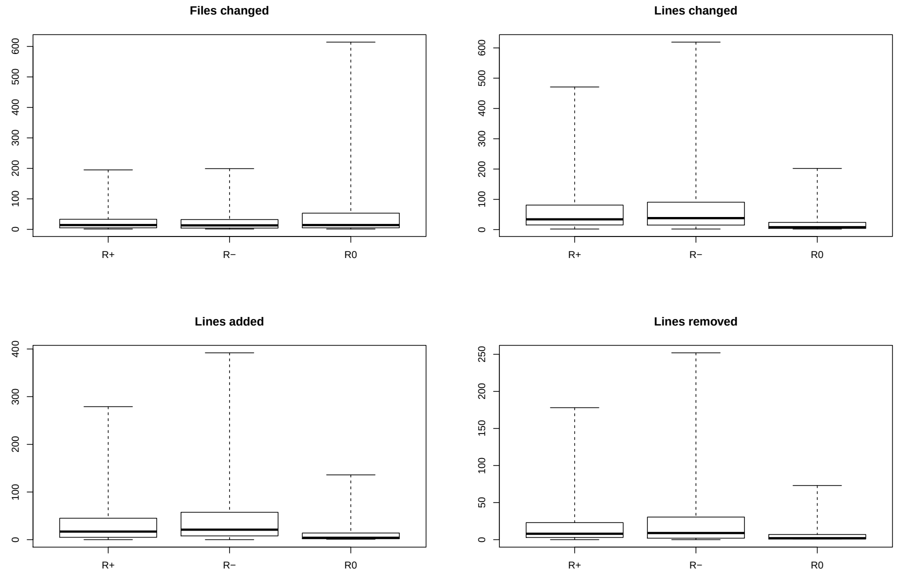


Figure 7.4: Adjusted boxplots of files changed, lines changed, lines added and lines removed for each group (without outliers).

be noticed that, in general, there is no significant difference when comparing R^+ and R^- in terms of modified lines, except for the comparison on the lines added (p-value $\simeq 0.017$), which is negligible anyway. The difference is always significant (p-value < 0.001) when comparing (i) R^+ and R^0, and (ii) R^- and R^0, even if with different values of Cliff's d.

Table 7.10 reports the effect size for the significant differences (by project) among the three compared groups in terms of total number of files changed and total number of lines changed; in Table 7.11 we report the same kind of comparison in terms of lines added and removed. In both the tables, we do not report values (cells marked as "//") for which the difference is not significant (p-value > 0.05). We could not compute the results for the project Hibernate Metamodel Generator because it only has transitions that do not change readability (*i.e., readable* → *readable* or *unreadable* → *unreadable*).

In general, we observed that changes that either improve or reduce code readability are different than changes that do not change readability in terms of number of modified lines (both added and removed). Specifically, it is more likely that code readability changes when many lines are added to or removed from the source file. Smaller changes, instead, are less likely to result in a readability modification.

The Case of ParSeq. We analyzed more in depth the project with a larger quantity of unreadable files to understand what caused this, *i.e.,* LinkedIn ParSeq. ParSeq is a framework that allows to simplify the writing of asynchronous code.

It is worth mentioning that, also given its nature, such a project makes heavy usage of the functional programming features introduced in Java 8 (*e.g.,* lambda expressions): if not used sparingly, such features may make the code difficult to read. This is the case of the class `Par2Task`[5]: the nested lambda expressions and the bad naming of the identifiers (*e.g.,* `p` or `o`) make such a class very hard to read. Then, we found that developers commonly used unpopular coding conventions: for example, a leading underscore is used for private fields (*e.g.,* "`_rand`"), which is more common in other programming languages, such as Python.

[5] Linkedin ParSeq, commit f3d9c: `https://git.io/JfiqM`

Table 7.10: Comparison on the subset with Wilcoxon rank-sum for various groups.

Project	**Files changed**			**Lines changed**		
	R^+ **vs.** R^-	R^+ **vs.** R^0	R^- **vs.** R^0	R^+ **vs.** R^-	R^+ **vs.** R^0	R^- **vs.** R^0
Ap. Beam	//	//	-0.425 (M)	-0.352 (M)	0.386 (M)	0.553 (L)
Ap. Cxf	//	//	//	//	0.523 (L)	0.677 (L)
Ap. Deltaspike	//	//	//	//	0.857 (L)	0.683 (L)
Ap. Falcon	//	//	//	//	0.493 (L)	//
Ap. Flink	0.257 (S)	//	-0.206 (S)	//	0.440 (M)	0.462 (M)
Ap. Hadoop	//	-0.367 (M)	-0.283 (S)	//	0.479 (L)	0.590 (L)
Ap. Inc.-S.	//	-0.277 (S)	//	//	0.471 (M)	0.470 (M)
Ap. Isis	//	//	//	//	0.676 (L)	0.399 (M)
Ap. Qpid-b.	//	//	//	//	0.600 (L)	0.608 (L)
Ap. Qpid	//	//	//	//	//	0.500 (L)
Ap. Tomcat	//	//	//	//	0.566 (L)	0.613 (L)
Fullcontact4j	//	0.861 (L)	//	//	//	//
Hib. Meta. Gen.	//	//	//	//	//	//
hlt-confdb	//	//	//	//	//	0.702 (L)
IGV	//	//	//	//	0.542 (L)	0.719 (L)
JBoss Modules	//	//	//	//	//	//
JBoss JBPM	//	//	//	//	0.873 (L)	//
NITHs	//	//	//	//	//	//
Nuxeo Runtime	//	//	//	//	//	//
OpenEngSB	-0.371 (M)	-0.477 (M)	-0.255 (S)	//	0.538 (L)	0.294 (S)
ParSeq	//	//	//	//	0.556 (L)	//
RxJava	//	//	//	//	0.518 (L)	0.527 (L)
SIB-dataportal	//	//	//	//	//	//
Undertow	//	//	//	//	0.636 (L)	0.710 (L)
Xnio	//	0.733 (L)	//	//	0.876 (L)	0.592 (L)

Table 7.11: Comparison on the subset with Wilcoxon rank-sum for various groups.

Project	Lines added			Lines removed		
	R^+ **vs.** R^-	R^+ **vs.** R^0	R^- **vs.** R^0	R^+ **vs.** R^-	R^+ **vs.** R^0	R^- **vs.** R^0
Ap. Beam	-0.547 (L)	0.237 (S)	0.596 (L)	//	0.515 (L)	0.313 (S)
Ap. Cxf	//	0.487 (L)	0.532 (L)	-0.326 (S)	//	0.408 (M)
Ap. Deltaspike	//	0.759 (L)	0.660 (L)	//	0.823 (L)	//
Ap. Falcon	//	//	//	//	//	//
Ap. Flink	//	0.323 (S)	0.432 (M)	//	0.435 (M)	0.265 (S)
Ap. Hadoop	//	0.494 (L)	0.544 (L)	//	//	//
Ap. Inc.-S.	//	0.469 (M)	0.355 (M)	//	0.280 (S)	0.425 (M)
Ap. Isis	//	0.386 (M)	0.348 (M)	0.436 (M)	0.645 (L)	0.302 (S)
Ap. Qpid-b.	-0.306 (S)	0.254 (S)	0.576 (L)	//	0.541 (L)	0.356 (M)
Ap. Qpid	//	//	0.520 (L)	//	//	//
Ap. Tomcat	-0.411 (M)	//	0.551 (L)	//	0.553 (L)	0.347 (M)
Fullcontact4j	//	//	//	//	//	//
Hib. Meta. Gen.	//	//	//	//	//	//
hlt-confdb	//	//	//	//	//	0.726 (L)
IGV	//	//	0.709 (L)	//	0.629 (L)	//
JBoss Modules	//	//	1.000 (L)	//	//	//
JBoss JBPM	//	0.738 (L)	//	//	0.651 (L)	//
NITHs	//	//	//	//	//	//
Nuxeo Runtime	//	//	//	//	//	//
OpenEngSB	//	0.497 (L)	0.322 (S)	//	//	//
ParSeq	//	//	//	//	//	//
RxJava	//	//	0.501 (L)	//	//	//
SIB-dataportal	//	//	//	//	//	//
Undertow	//	0.421 (M)	0.528 (L)	//	0.504 (L)	0.309 (S)
Xnio	//	0.896 (L)	//	//	//	//

It can be noticed that such an uncommon convention (leading underscore) is used also for method names, as it is possible to observe in the previously mentioned class `Par2Task`. Specifically, let us consider this line:

```
return map(tuple -> f.apply(tuple._1(), tuple._2()));
```

In this case, there is a call to two public methods that contain a leading underscore. It is worth noting that the presence of such methods does not only negatively affect the readability of the class that contains them, but also the readability of the classes that use them, such as the one previously mentioned.

Looking at the evolution of files in ParSeq, we noticed that, in general, developers first try to quickly implement features and then they optionally refine the classes and improve their quality, including readability. For example, our dataset contains 25 readability transitions for the class `BatchingStrategy`: such a class was created unreadable and it remained unreadable for 14 transitions; then, a commit adds a transition *unreadable* → *readable* and, finally, this file remains readable for other 9 transitions. In the second commit[6], the file completely misses Javadoc comments. Later, in commit `fc27e`[7] developers added Javadoc comments to the class and, finally, in commit `8d7a3`[8], the file became readable (*i.e., unreadable* → *readable*), thanks to other small improvements to the code structure. However, such an approach, as evidence shows, does not always work, because only a small percentage of unreadable file of such a project become readable (*i.e.,* 0.8%). All the others remain unreadable.

For example, this is the case of class `ClassifierDriver`. This file is involved in 3 transitions. One of the commits that modify such a file is `a9758`[9]: even if such a change is specifically aimed at improving the readability by unifying code formatting, this is not enough to make the file readable.

Summary of RQ_3. Code readability of an individual file rarely changes during the evolution of a project.

[6] Linkedin ParSeq, commit `ce2ae`: `https://git.io/Jf6jp`
[7] Linkedin ParSeq, commit `fc27e`: `https://git.io/Jfim5`
[8] Linkedin ParSeq, commit `8d7a3`: `https://git.io/JfwPs`
[9] Linkedin ParSeq, commit `a9758`: `https://git.io/JfwXR`

RQ_4: How and Why Does Code Readability Change?

We found that 57 files had extreme readability scores in their history (*i.e.,* both < 0.25 and > 0.75). For such files, in total, we considered 82 commits and commit sequences that changed code readability. We found 16 commits and 1 commit sequence for which we did not agree with the output of the tool (*i.e.,* the tool misclassified code readability). We excluded such cases and, therefore, we considered 65 commits and commit sequences.

We found that most of the readability changes (82.0% of the cases) were caused by *adaptive* changes, 14.7% of them were caused by *perfective* changes, while only 3.3% of them happened because of *corrective* changes. As it could be expected, all the perfective changes improved code readability: it is interesting, however, that even perfective changes not explicitly aimed at improving code readability may have that as a side-effect. We also found that both of the two corrective changes we analyzed improved code readability: it is well known that, sometimes, developers refactor code before fixing bugs; it is less expected, instead, that corrective changes usually improve code readability. While such findings may seem expected, to the best of our knowledge this is the first piece of empirical work that relates the type of changes to code readability. This kind of evidence is important since, as previous work shows [170], developers' perception may not be always well captured by code metrics: it could happen that developers think they are improving some aspects of source code, while this is not case, or, on the other hand, it could happen that code metrics are not sufficiently sophisticated to capture the improvement. We report below some interesting cases we found.

Refactoring improved readability. Six perfective changes out of nine are from Apache Incubator-Skywalking. The developers decided to refactor a part of the system involving module installers. This operation was aimed at reducing the complexity of single installers, moving it to super classes. The result of this refactoring, indeed, resulted in a profound improvement in code readability for some of the installers which contained more articulated code. It is worth noting that such changes were not explicitly intended to improve the readability: this was a side effect of refactoring.

Bug fixing improved readability. A clear example of corrective change that results in higher readability comes from Apache Deltaspike. The developers fixed parts of the project and, in doing so, also added a comment in a small class. This comment was aimed at clarifying the purpose of a line. This addition increased both the number of comments and the consistency between comments and identifiers. This modification also increased the readability of the whole class.

Adaptive changes. In Apache Incubator-Skywalking, we found 16 cases in which code became unreadable. Looking at the code, we found that developers first implemented empty versions of some methods (*e.g.,* containing just `return 0;`). Such versions were clearly readable. After this first phase, they actually implemented such methods and, in doing so, they incidentally introduced unreadable code. Therefore, even if in this case we face a change in readability, this happens just because the developers created the classes in two steps. We found similar examples also in other projects, such as Apache Falcon. In other cases, we found that code readability decreased because simple methods were removed. In fact, we compute the readability of a class as the mean readability of the methods that compose it: if a class contains both readable and unreadable methods, the code readability decreases when the number of readable methods decreases. This happened, for example, in a commit[10] from Apache Tomcat in which 13 very simple methods were removed from the class `StackMapTableEntry`, in a commit[11] from Apache Beam in which five empty methods were deleted from the class `FlinkStateInternalsTest` and in a commit[12] Apache Qpid-broker-j in which two empty methods with documentation were removed from class `Refresh`. Similarly, most of the adaptive changes that resulted in a code readability improvement were incidental. In six cases, unreadable code was removed or commented. This happened, for example, in the test `HandlerComparatorTest` in Apache Deltaspike[13].

The results of our qualitative analysis agree with what was already observed for other bad pracites (*e.g.,* code smells) [44, 65, 224, 208, 141, 241]: changes in readability are mostly done unintentionally.

[10] Apache Tomcat, commit `7d99e`: `https://git.io/fjLRw`

[11] Apache Beam, commit `7126f`: `https://git.io/fjLR9`

[12] Apache Qpid-broker-j, commit `2d90e`: `https://git.io/fjLRl`

[13] Apache Deltaspike, commit `36861`: `https://git.io/fhHpp`

Why does Code Readability Change? We looked more in depth into the causes of the changes in code readability. We did this in terms of *structural*, *visual*, and *textual* aspects, *i.e.,* the ones used in the state of the art to predict code readability.

We found 34 cases in which code readability *decreases* because of *structural* aspects. The most common cause is the introduction of long lines of code (17 cases): for example, in a commit[14] from Apache Incubator Skywalking, it is possible to find, among the other possible problems, that the longest line of code has 116 characters; the Java guidelines[15] suggest to make lines shorter than 100 or even 80 characters.

We also found 19 cases in which the problem was the introduction of high levels of nesting (*e.g.,* `if-else` statements or loops). For example, in a commit[16] from Apache Incubator Skywalking, it is possible to find in the class `SegmentH2DAO` the introduction of a `try` block in two nested `if` statements, nested in another `try` block. Other common problems include higher number of parentheses and complex arithmetic expressions (*e.g.,* higher number of operators). In 27 cases, code readability *increased* because of improvements of some *structural* aspects. Conversely to what happened for readability decrease, in these cases nested blocks mostly disappeared: in a commit[17] from Apache Falcon, the class `LateDataUtils` was refactored by extracting a long and complex instruction and putting it in a method on its own. This helped increasing the whole readability of the class.

We found 26 cases in which readability *decreased* because of changes in *visual* aspects. In 25 cases it is possible to see that indentation was not properly used. This happened, for example, in a commit[18] from Apache Incubator Skywalking: the class `ApplicationH2DAO` does not have proper indentation; besides, a line starts with a "`;`" which was, most likely, not intended to be there. Conversely, we found 8 cases in which readability *improved* as a consequence of changes in *visual*

[14] Apache Incubator Skywalking, commit `ca90b`: `https://git.io/JeB5z`

[15] The ones by Oracle, `http://www.oracle.com/technetwork/java/codeconventions-150003.pdf`, and the ones by Google, `https://google.github.io/styleguide/javaguide.html`

[16] Apache Incubator Skywalking, commit `d4333`: `https://git.io/JeB9a`

[17] Apache Falcon, commit `3769e`: `https://git.io/JeBbW`

[18] Apache Incubator Skywalking, commit `bc38a`: `https://git.io/JeB50`

aspects. For example, in a commit[19] from Reactive RxJava, in which in the class `OperatorMulticast` is added code with a good indentation.

We found six cases in which readability *decreased* because of *textual* aspects, most of which regarding problems in the names of the identifiers (*e.g.,* wrong word splitting or abbreviations). For example, this occurred in a commit[20] from Apache Tomcat: simple methods were removed and a remaining method, *i.e.,* `toString`, was unreadable also because of the presence of many occurrences of the identifier `buf`, short for "string buffer". In 15 cases readability *increased* because of improvements in *textual* aspects instead. This happened because the developers added comments, improved bad identifiers or the textual cohesion. For example, this is the case of a commit[21] from Apache Hadoop in which the method `testDynamicLogLevel`, that implemented many concepts (possibly, an *eager test*), was divided in many methods (*e.g.,* `testLogLevelByHttp`) with a higher textual cohesion.

Summary of RQ_4. We observed that (i) big code changes in which new code is added are the most prone to reduce code readability, and (ii) readability is increased/decreased mostly unintentionally.

Discussion

The main finding of our study is that code readability rarely changes. A first hint towards this finding could be found in the survey presented in Chapter 6, in which most of the developers declared that, according to their experience, readability changes only in less than 25% of the cases. Our empirical results confirm this, showing that such a percentage is, actually, very low (always lower than 6.3%, for the projects we studied). Moreover, our results show that it is more likely that developers make unreadable files readable than the opposite. Another interesting phenomenon we observed is that, even if it is generally a minority, unreadable code tends to stay that way: on average, about 10% of the files of a project are created unreadable and remain unreadable, with some outliers, such as ParSeq, for which the percentage of unreadable files is very high (33.9%).

[19] Reactive RxJava, commit `0499c`: `https://git.io/JeREW`

[20] Apache Tomcat, commit `7d99e`: `https://git.io/fjLR2`

[21] Apache Hadoop, commit `34cc2`: `https://git.io/JeRZK`

The risk of introducing unreadable code is higher when developers introduce new code in a project: when developers create new files, there is a relatively high probability that such files are unreadable. On the other hand, readable files rarely become unreadable (the estimated probability is always lower than 4%). In our qualitative analysis, we found that, when this happens, it is because of adaptive changes. Something similar was already observed in the case of introduction of code smells [224] and dependencies on unstable APIs [60]: both code smells and dependencies on unstable APIs are mostly present from the beginning and are rarely introduced during code evolution.

Based on our empirical results, we define several guidelines to help developers keeping the quantity of unreadable code low. It is worth noting that some of them agree with what is already known to be beneficial to avoid the introduction of problems (*e.g.,* bugs): we still think it is interesting to know that such guidelines also help avoiding the introduction of unreadable code.

- **Do not write code that should be improved in the future.** One of the clearest evidence we obtained from the results of our study is that readability hardly changes during software evolution. This means that developers should not underestimate readability when writing code since it is unlikely that unreadable code later becomes readable. This can happen either because developers do not think it is worth spending time making code more readable (they may have more important tasks to complete) or because the readability may become so low that it is very hard to recover such situations (like in the previously analyzed ParSeq). Writing code that should be improved in the future can be seen as introducing technical debt that should be resolved later as of Cunningham [75]: at the very least such technical debt should be either explicitly admitted by developers [182], or flagged by specialized tools [240].

- **Reviews for code readability should be focused on new files and big changes.** Unreadable code is introduced mostly when new files/classes are created; when a class is unreadable, it will most likely remain that way during the entire project evolution. It would be a good practice to conduct code reviews specifically aimed at checking the readability of the new classes suspected of being unreadable. Our analyses also show that code changes bigger than usual may result in a change in code readability: therefore, big changes should also receive special attention. Code readability estimation tools could be used to reduce the number of classes to review (*e.g.,* limiting the review to potentially unreadable classes). It is worth noting that such a guideline does not replace other general guidelines on code review aimed at checking the presence of bugs, for which even small changes may be detrimental.

- **Prefer small incremental changes.** It is well known that commits should be small and consistent. We found that big (non-perfective) code changes are dangerous for code readability as well as new file creation operations. As described by Graves *et al.* [102], the introduction of new files could also more likely include bugs. Even if the probability of making a readable file unreadable is low, it is still worth reviewing such modifications since unreadable files would most likely remain unreadable. This guideline agrees with previous findings: for example, Purushothaman *et al.* [184] showed that most of the small code changes do not result in the introduction of defects in the software. When big code changes are necessary, performing code reviews aimed at checking the code readability could help reducing the risk of readability deterioration. Furthermore, Fowler *et al.* [98] and Duvall *et al.* [81] support commit with small changes in the CI guidelines. Zhao *et al.* [244] have found that this guideline is followed only to some extent, with large differences between projects in term of adherence to this guideline. In a recent study by Ebert *et al.* [83] on the confusion in code reviews, long and complex changes have been repeatedly reported as a reason for confusion.

- **Refactor code when possible.** Refactoring operations are done to improve code maintainability. Our results show that it is beneficial also for code readability. Surprisingly, we observed improvements in code readability even when refactoring was not directly done with this aim. The following guideline reminds the opposition between floss-refactoring and root canal refactoring [157]. Developers already prefer the floss-refactoring (*i.e.,* frequent refactoring steps interleaved with their regular activities). Consequently, in projects with a big number of unreadable files there may be a need of perform more refactoring operations. It is worth noting that this guideline is not in contrast with the first one: readability should not be underestimated during development, but perfective changes are still beneficial, above all if the readability of some classes is borderline.

- **Carefully control the interface of the most used classes.** The design of the classes most used in a project may have a strong impact on the readability of other classes. For example, if a popular class contains a method with a unconventional name (like the previously mentioned `_1` in ParSeq), the readability of classes using such a method may be negatively impacted. For this reason, the public methods exposed by classes, especially the most used ones, should be carefully decided and kept up-to-date (*e.g.,* if the purpose of the method is slightly changed) during the evolution of a project.

We also delineate possible future research directions in the field of code readability prediction:

- **Define a readability-transition prediction model.** Our results show that readability prediction models aimed at predicting readability of single versions achieve modest accuracy on predicting readability evolution transitions: the state-of-the-art readability prediction tool that achieves the highest accuracy on single file versions allows to achieve only a 64.5% precision on *readable* → *unreadable* transitions. This is because the problem at hand is different and some features (*e.g.,* number of changed lines or author's characteristics) are obviously ignored by such models. A new readability-transition prediction model would be useful for developers, since changes that make readable code unreadable are generally rare and hard to catch manually; also, when a transition of this type happens, it is very unlikely that an opposite transition is introduced since, in general, files with low readability rarely become readable.

- **Improving the readability-transition model.** Our readability-transition model is based on a binary classification of code readability. However, code readability models do not only provide the binary classification, but also the probability that such a classification is correct (*i.e.,* the output score is in the range [0, 1]). Future research could be aimed at finding the best way of including such an information in the model to improve the accuracy through which it describes the evolution of code readability. Besides, the use of non-time-homogeneous Markov chains could be investigated to take into account how the evolution of a project changes the transition probabilities in the model.

- **Experiment the use of readability tools in CI.** A tool that automatically detects transitions that make code unreadable may be useful in practice, and it could be integrated in Continuous Integration pipelines to automatically warn developers when they make readability reducing changes in commits, before other developers are involved in the code review process. However, before this can happen, it would be necessary to understand to what extent developers would benefit from such tools: would such tools help them keeping the code readable? Would they find warning useful or bothering? Empirical evidence is needed to find the answer to such questions.

7.5 Threats to Validity

In this section we analyze and discuss the threats that could affect the validity of the results achieved. We describe construct validity, internal validity and external validity.

Construct Validity. The dataset is constructed by mining software repositories using Git from GitHub. GitHub is widely used in software engineering research. However, there are many possible perils in automatically extracting information from such sources [53, 115]. We made sure that we excluded personal/toy projects and repositories not used for software development (see the work of Munaiah *et al.* [154] for a more advanced treatment of this subject). We did not explicitly check if the repositories we used were actively developed: we singularly analyzed each project. Therefore we believe that the lack of recent activity for some of them does not affect the validity of our results. We avoided possible problems related to GitHub APIs (*e.g.,* the fact that some APIs do not expose all the data) by cloning and locally analyzing the Git repositories.

The model we used to compute the readability of the files in our dataset can wrongly classify a readable file as unreadable and vice versa. We limited this threat by using the model that in literature is reported as the one with the highest accuracy [199]. Furthermore, Git allows developers to rewrite the history: there is a risk that this could have affected our analysis. Another possible threat is represented by renaming/moving operations: Git sometimes does not correctly detect such operations and it interprets them as combinations of file removal and addition instead, in such cases we may have wrongly used the *non-existing* instead of *other-name* as initial state.

We operationalized the readability at class-level as the arithmetic mean of the readability computed at method-level. It may be argued that other aggregation techniques would have provided more reliable estimation of class-level readability [228]: arithmetic mean does not work well when a class is composed by many readable methods (*e.g.,* getters and setters) and a single unreadable method [228]. We list below other measures of central tendency and discuss their advantages and disadvantages.

- **Minimum**: this would have been useful in the scenario described before (many small readable methods and a single unreadable method). However, the probability of making a mistake would have been much higher in the average case: the probability of correctly classifying C as readable or unreadable would have been equal to $P_{correct}(C) = \Pi_{i=1}^{n} P_{correct}(C_i)$. Assuming that the probability of correctly classifying a method is constant (*i.e.,* ∼84%, according to the study of [199]), this value can be approximated to 0.84^n. In other words, in large classes, using the minimum could have negatively affected the overall accuracy of the classifier, since it would have been sufficient to wrongly classify a single method as unreadable to make a mistake for the whole class. For example, for a class with 10 methods the probability of correctly classifying C would have dropped to about 17.5%.

- **Median**: this aggregation is preferred over mean for skewed distributions, since it is more robust and it allows to ignore extreme values (possible outliers). In this context, however, this was not the best choice since extreme values are the most relevant ones, *i.e.,* the ones on which the classifier is more confident. Moreover, this kind of aggregation would have not solved the problem of the arithmetic mean (many readable methods and a single unreadable method).

- **Weighted mean**: while this aggregation could have been suitable for handling cases in which there are many getters and setters and a single unreadable long method, it is worth noting that unreadable methods are not necessarily longer than readable ones. Instead, there could be very short methods that make the class unreadable. Such methods would have had a possibly smaller weight. Moreover, such an aggregation would have forced us to assume that long methods matter to developers more than short methods: we decided not make such a strong assumption and, therefore, not to use this aggregation.

Finally, to define our readability evolution model, we assumed that readability is not a cause for deletion of source files and, therefore, we did not track such operations. This assumption does not affect the results of our study; however, future work specifically aimed at finding the causes of class deletions may be done to investigate more in-depth if low readability has a role in this.

Internal Validity. Because of the errors made by the readability classifier we used, it is possible that the frequency of some transitions is larger/smaller than it is in the reality. To limit the impact of the threat that small changes around 0.5 result in a new transition in our dataset, we excluded the borderline values (between ∼0.416 and ∼0.600, based on the results of RQ_1). However, this caused the exclusion of about 20% of the transitions we recorded: there is a risk that some of such transitions were not false positives and, therefore, that we missed meaningful transitions in our subsequent analyses.

To answer RQ_2, the raters needed to state their agreement with the evaluations automatically performed by the tool. The raters might have inclined to agree with the tool. To reduce the impact of such a threat, we made sure that at least two authors independently rated each occurrence. Besides, in the results of RQ_2, we showed also that the tool is not as accurate in classifying transitions as when it is used on single file versions. We limited this threat using bootstrapping for estimating the probabilities and we reported the 90% confidence intervals for each probability.

The model we defined describes the probability that a file modification results in a change of code readability. Since we perform our analysis at commit-level, there is the risk that small commits may gradually erode code readability until the state of a file changes with a single commit. In other words, the probabilities we report could be biased by the fact that most of the changes are small. Other granularity levels (*e.g.,* release-level) would have avoided this threat, but they would have not allowed us to understand what happens more in details. To limit this threat, we also report the number of files that change readability at least once (*i.e.,* regardless of the number of commits).

We used a time-homogeneous Markov chain to describe the readability evolution process. In other words, we assumed that the probabilities of state transitions do not change in time. Such assumption could not always hold: for example, the probability of introducing *unreadable* files may be higher when many new developers start contributing to the project (as we observed in the results of RQ_3). Future studies should consider also the usage of generic discrete-time Markov chains.

External Validity. The conclusions of our study may be limited to the 25 projects we considered in our experiment. We randomly selected such systems considering only the ones with a reasonably big history and big enough that readability monitoring may matter for their developers. Besides, we considered only open source Java projects: the results may not be necessarily generalize to industrial software or software written in other programming languages.

Furthermore, in the selection of open source Java projects, we selected well-known projects that are still actively developed and this could be have biased our results (survivor bias). Abandoned or failed projects could have had very different characteristics: for example, it would have been possible to observe a decline in terms of code readability at the end of the history of a project.

Another threat to validity regard results of the study. Despite the results achieved (the transition *non-existing* → *readable* has 88.2% of precision and 68.2% of recall, while the transition *unreadable* → *readable* has 90.5% of recall and the

57.6% of precision) are almost in line with results obtained in other contexts (*i.e.,* in the recovery of traceability links or the defect prediction), we cannot say that these values of precision and recall are sufficient for a reliable analysis of readability evolution. To understand if these values are sufficient, there is the need of studies designed ad-hoc to evaluate the effectiveness of the proposed approach.

7.6 Final Remarks

Readability is one of the most desirable characteristics of source code: if code is hard to read, it is likely that it will cost a developer more effort to understand it during maintenance.

In this chapter, we introduced a descriptive model for the evolution of code readability at file level. The goal of our large empirical study is to understand how and why code readability changes during software evolution. We considered the history of 25 projects, for a total of ∼83K commits.

The results of our study show that:

- using readability prediction tools for transitions reduces their accuracy since such tool are designed to work on single snapshots;
- only in a minorty of the cases (< 5%) the code readability of individual files changes during the evolution of a project: when a file is created *unreadable* it most likely remains *unreadable*;
- adaptive changes and, specifically, big commits in which new code is introduced are the most prone to reduce code readability;
- code readability increases and decreases mostly unintentionally.

The empirical results obtained allow us to distill several lessons learned for developers:

- readability-oriented code reviews should be mostly focus on new classes;
- small incremental commits should be preferred to reduce the risk of readability reduction;

- refactoring should be encouraged, when possible, since it has a positive effect on readability.

Part III

Code Readability & Cognitive Human Factors

"Experience without theory is blind,
but theory without experience is mere intellectual play."
Kant, Immanuel

CHAPTER 8

Can Cognitive Human Factors Affect Code Quality?

Contents

8.1 Introduction

In Part II we obtain important findings. Specifically, from Chapter 5, we can derive that developers perform refactoring operations to improve code readability. With the study in Chapter 6 we confirm these results because developers declare that code readability is very important in their writing code activities. From their declarations, in Chapter 7, we deepen dynamics related to the evolution of the code readability in open source projects. From this study, we obtain that the readability of a code changes rarely. Indeed, a code is created readable and remains readable because it undergoes small variations on code readability, which do not allow the file to change its readability status to unreadable. This happens also when a file is created unreadable.

At light of these results, we can conclude that the evolution does not deteriorate code readability. If the evolution does not deteriorate code readability, we conjecture that specific personal characteristics of developers can influence the quality of a software component, *i.e.,* code readability. Software development and evolution are complex activities involving writing and modification of the source code. Developers are continuously presented with coding tasks in their daily activities, such as implementing a new feature or fixing bugs. Being able to estimate the time required to complete a coding task and predict its internal and external quality (*e.g.,* correctness of coding tasks) would allow to better allocate the effort of a development team to optimize the final outcome in terms of time and quality. As for the correctness of source code, *defect prediction* is an active field of software engineering research aiming at identifying characteristics of the software systems and projects (*e.g.,* code quality) [245, 109, 185, 135, 219]. In the last decade, researchers have started including in such models factors related to the developers, their attention focus [180] or the scattering of changes they performed [77]. The majority of existing defect prediction models, however, completely neglect the role that *cognitive human aspects* can have on the final outcome of a coding task.

In this regard, the literature in psychology reports that cognitive human aspects correlate with different outcomes in human activities, even complex ones. More specifically, attention- and memory-related factors are shown to be related to activities that require problem-solving. For example, several studies [233, 178, 85] used attention-related factors to predict the outcome in driving, while others have shown that both attention and memory correlate with the performance in mathematics [159, 234, 172].

In this chapter, we theorize that cognitive human aspects, and, specifically, attention- and memory-related ones, play a role in explaining the outcome of coding tasks in terms of *time* required to complete a task, and quality of the final solution, here measured in terms of *correctness* (external quality) and *code readability* (internal quality). As for attention, we focus on three factors: *alerting* (*i.e.,* the ability of gaining and maintaining an alert state), *orienting* (*i.e.,* the ability of selecting information from sensory input), and *executive control* (*i.e.,* the ability of dealing with conflict among responses); as for memory, we consider *immediate recall* (*i.e.,* the ability of recalling information acquired in the very short term), and *working memory* (used for elaborating problem-solving strategies [40, 205]). We decided to focus on these cognitive human aspects because Peitek *et al.* [173], Siegmund *et al.* [207] and Krueger *et al.* [126] demonstrated that during coding activities (*i.e.,* program comprehension and code writing) there are specific neural activations related to *attention* and *working memory.* To do this, authors use the functional magnetic resonance imaging (fMRI) brain scan to measure the neural activities during the coding activities.

If we prove our theory, we envision two future directions. First, researchers would be able to define personalized defect prediction models. Second, specific cognitive training sessions [161] for developers could be devised, aimed at improving the most important attention- and memory-related factors. In this way, developers can obtain a specific preparation on these factors to implement at best their programming tasks. For example, this would allow them to write better code in less time. To test our theory, we conducted a controlled experiment for answering to the following research questions:

- RQ_1: *To what extent do attention and memory have an impact on the correctness of the solution of coding tasks?*
- RQ_2: *To what extent do attention and memory have an impact on the time needed to complete coding tasks?*
- RQ_3: *To what extent do attention and memory have an impact on readability of the solution of coding tasks?*

We involved 32 participants with different backgrounds and asked each of them to complete two bug fixing and two feature implementation tasks. Since we were interested in measuring the influence of cognitive human aspects indipendently from the context in which such tasks could have been completed (*e.g.,* a specific software system), we provided developers with stand-alone problems (*i.e.,* not requiring the knowledge of other software components) that could be solved in the time we allocated for tasks (30 minutes). We measure attention- and memory-related factors in isolation to have the personal characteristics of developers before of the programming tasks given that cognitive human aspects cannot be measured simultaneously with the programming tasks. We used state-of-the-art psychometric tests to measure the attention- and memory-related factors. We test our theory to verify whether there are relations between such variables and two variables related to the outcome of the task: *correctness* (percentage of test cases passed), *time* (minutes required to complete the task) and *readability* (computed using the metric by Scalabrino *et al.* [199]).

Our results show that the correlation was not statistically significant between all the attention- and memory-related factors and the one dependent variable (*i.e., correctness*), both when considered alone and when combined in regression models. The only developer-related variable that we found to be significantly related to *correctness* is *programming experience.* On the other hand, we observe a statistically significant relationship between an attention-related factors (alerting, p-value = 0.046 , and orienting, p-value = 0.031) and *code readability.* In addition, there is a significant correlation between a memory-related factor, *immediate recall*, p-value = 0.013, and *time.* While experiments conducted with different and larger samples are required to further corroborate our findings, our results provide a clear message to future researchers interested in this field: *Attention-*

and memory-related factors should be investigated with other external quality, and other cognitive human aspects should be investigated considering also other external quality. Also, programming experience should always be taken into account, since it is significantly related to both time and correctness.

The remainder of our chapter is organized as follows. In Section 8.2 we present our theory. In Section 8.3 we report the design of our study, while in Section 8.4 we show the results of our analysis. Section 8.5 describes the threats to validity, and Section 8.6 concludes this chapter.

8.2 Cognitive Factors and Software Development Tasks

We theorize that attention- and memory-related factors allow to explain the *correctness* of a task and the *time* needed to complete it. In the following section, we provide more details on our theory.

8.2.1 Attention

Attention is part of executive functions, as planning, sequencing and cognitive flexibility [74]. As explained in Section 8.1, attention can be controlled through three key aspects, *i.e.,* alerting, orienting and executive control, that provide the reactivity to a specific event or stimulus [178].

As reported by Peitek *et al.* [173], Siegmund *et al.* [207] and Krueger *et al.* [126], the attention is a neural activation both for the code writing [126] and for the program comprehension [173, 207]. Thus, attention is an active part of the right emisphere when the developer writes and comprehend the code. Given these results [173, 126], the attention is important both for the implementation of a new feature and for the fixing of a bug. We expect that higher efficiency in the *alerting* network would positively affect the outcome (both *correctness* and *time*) of coding tasks, because a developer would manage alert situation also in stressful situations (*i.e.,* a short deadline). For example, if developers has to release a functionality and the deadline is one hour later, they find a bug a few minutes before the deadline, they has to fix it in the minimum time possible. To fix a bug quickly developers cannot be distracted and they needs to maintain an alert state. We conjecture that higher efficiency in the *orienting* network would allow to achieve better results (again, for both the dependent variables) because developers would be able to better focus on a specific area of the code. In addition, developers would be able to apply coding conventions in their writing activities and verify that the code readability is positive or negative. Thus, we hypothesize that higher efficiency in the *executive control* network would allow developers to complete tasks better (*correctness*), more quickly (*time*) and to implement a code readable (*readability*) because an individual with a high executive control is able to better isolate conflicting stimuli (*e.g.,* the ones from the environment from the ones of the task) and, thus, focus more on the task at hand.

8.2.2 Memory

In the studies of Peitek *et al.* [173], Siegmund *et al.* [206] and Krueger *et al.* [126], memory is another neural activation during the code writing and the program comprehension. The *immediate recall* is part of the episodic memory and this memory contains a good part of past events [201]. We conjecture that developers with a good ability to recall might be able to better re-use patterns of solutions applied in the past in similar situations. For example, developers have to fix a bug and they have fixed a similar bug in past; thus, developers could remember the related fix. This would result in saving *time* and, possibly, in higher chances of spending more time ensuring that the solution provided is correct (*i.e.,* in higher *correctness*) and also of writing readable code (*i.e.,* high *readability*). The human *working memory* provides a temporary storage of information necessary for other cognitive tasks (*e.g.,* reading or problem-solving) [40]. Specifically, developers might use their working memory to elaborate solution strategies [205]. Furthermore, developers with a strong working memory can also manipulate numbers and words to implement a readable code [173]. We conjecture that a good working memory allows developers to complete a coding task both more correctly and more efficiently (*i.e.,* shorter *time* needed).

8.3 Design of the Study

The *goal* of the study is to verify to what extent attention and memory have an impact on how developers complete coding tasks. The *perspective* is of researchers that aim at measuring the influence of cognitive human factors on developers' performance (*time* needed to complete a task and *correctness* and *readability* of the solution).

Specifically, we formulate and address the following research questions:

- RQ_1: *To what extent do attention and memory have an impact on the correctness of the solution of coding tasks?*

- RQ_2: *To what extent do attention and memory have an impact on the time needed to complete coding tasks?*

- RQ_3: *To what extent do attention and memory have an impact on readability of the solution of coding tasks?*

To answer our research questions, we conducted a controlled experiment in which we collect measurements for both the dependent (*i.e., correctness*, *time*, and *readability*) and the independent variables (attention- and memory- related factors) in which we are interested. Besides such factors, we also include other independent variables that are commonly associated with the outcome of coding tasks, *i.e., task type*, *task difficulty*, and developer's *programming experience.*

Task Type. Some developers might find easier to implement a new feature from scratch, because they do not need to deal with code written by other developers; some others, instead, might find easier starting from a partial solution and modify it to make it work as intended. Therefore, task type may be naturally associated with the *correctness* of the solution. We also assume that task type explains variance in the *time* needed to complete a task: we expect that *bug fixing* tasks generally require more time since most of the code is written; however, this might strongly depend on the developer, as previously explained [188].

Task Difficulty. Difficulty is naturally associated both with the *correctness* of the solution: intuitively, it is more likely that a developer writes bug-free code when presented with an easy task; similarly, we can assume that easier tasks take

```
public boolean match(String word, String pattern) {
  Map<Character, Character> m1 = new HashMap();
  ~~Map<Character, Character> m2 = new HashMap();~~

  for (int i = 0; i < word.length(); ++i) {
    char w = word.charAt(i);
    char p = pattern.charAt(i);
    if (!m1.containsKey(w)) m1.put(w, p);
    if (!~~m2~~m1.containsKey(p)) ~~m2~~m1.put(p, w);
    if (m1.get(w) != p || ~~m2~~m1.get(p) != w)
    return false;
  }

  return true;
}
```

Figure 8.1: Example of an injected bug in the easy task. We decide to remove the hash map m2 from the originally correct solution.

generally less time to be completed. While difficulty is somewhat a subjective concept and it might depend on the knowledge and experience of the developer, there are some objective features that make some tasks more difficult than others. For example, everything else being equal, fixing a bug that involves one line of code is inherently easier than fixing a bug involving multiple lines of code, especially if these lines of code are in various part of the system [188, 242].

Experience. It is a common understanding that the *programming experience* plays a significant role in different kinds of software engineering tasks [190, 191, 206]. We assume that more experienced developers can complete tasks more correctly (higher *correctness*) and more quickly (lower *time*). We measure the programming experience by asking developers to report the number of years of experience (*i.e.,* from their first programming task).

In the following, we describe (i) the context of our experiment, *i.e.,* the participants and the tasks, (ii) the procedure we used to collect the data, *i.e.,* how we run the experiment, and (iii) how we analyzed the collected data to answer our research questions.

8.3.1 Context Selection

The context of our study is composed of *objects*, *i.e.,* coding tasks, and *subjects*, *i.e.,* software developers. To select the tasks we used in our study, we relied on

LeetCode,[1] an online platform commonly used to exercise coding problems. The platform proposes a wide range of problems that can be solved by users in many programming languages. A problem in LeetCode is usually composed by (i) a description of the problem, and (ii) at least an example of input and expected output. The developer can, then, implement a solution, manually test it, or submit it. In the latter case, LeetCode runs a test suite to check if the solution is correct. To select the participants, instead, we used convenience sampling, and we invited people within the personal network of the authors and through student channels. We provide below more details about the selection of tasks and participants.

Task Selection. The two aspects we controlled for in our experiment in terms of task are the difficulty and the task type. As for the difficulty, we aimed at having both *easy* and *hard* tasks. As for the task type, we wanted to cover two categories of tasks typically performed during software development and maintenance: *feature implementation* and *bug fixing.* In the first category, developers are requested to implement code from scratch, while, in the second one, they are provided with a partially correct solution that they need to modify. We defined a total of 4 tasks to cover all the possible combinations of task difficulty and type.

To define the tasks, as a first step, we started from the pool of all the problems available in LeetCode, and filtered them based on the difficulty tags. The difficulty tag on LeetCode is manually assigned by the person who originally proposed the problem. We selected two separated pools of problems: *easy* ones and *hard* ones, discarding problems of *medium* difficulty. We arbitrarily picked two easy tasks and two hard tasks from such pools. We verified that for all the problems we selected it was possible to submit a solution in Java since we planned to ask developers to complete the tasks using Java.

The definition of the *feature implementation* tasks starting from the LeetCode problems was straightforward: We simply provided the participants with the problem description and we asked them to implement a solution from scratch. To define the *bug fixing* tasks, instead, we needed to provide a buggy solution that they would have fixed. To define the buggy solution, we started from the correct solution provided by LeetCode itself. Then, we manually injected bugs in such a

[1] https://leetcode.com/

Table 8.1: Tasks selected for the experiment. 🐞 and 🥚 indicate task types (*bug fixing* and *feature implementation*), while ︾ and ︽ indicate task difficulties (*easy* and *hard*).

Problem	**Type**	**Difficulty**[2]	**#Test Cases**[3]
Find and Replace Pattern [13]	🐞	︾	47
Sort the Matrix Diagonally [15]	🐞	︽	15
Duplicate Zeros [12]	🥚	︾	30
Regular Expression Matching [14]	🥚	︽	352

solution. The type of modification we made to the code depended on the difficulty of the task. Specifically, we introduced more articulated bugs in difficult task. We report an example of injected bug in Figure 8.1. We report in Table 8.1 the main features of the four tasks we defined. Specifically, for each task we report the difficulty level of tasks, *i.e.,* the value reported when we selected the task. The difficulty of both the bug fixing tasks was later changed to "Medium". In addition, for each task we report the number of test cases on LeetCode. Participants did not have access to these test cases and we internally used them to check the *correctness* of the solutions they provided.

Participants Selection. To define the number of data points we would have needed to observe variations in the dependent variables due to the factors we studied, we ran a power analysis for linear regression.[4] To identify a model with $f^2 \simeq 0.15$ (*i.e.,* $R^2 \simeq 0.13$) using 8 predictors (more on the model later) with a 80% power, we needed at least 109 data points. Since each participant produces four data points, one for each task, we needed at least 28 participants. As output of the recruiting phase, we involved 32 participants, distributed as follows: 18 bachelor students, 9 master students, 2 Ph.D. students, and 3 practitioners.

Table 8.2 reports some demographic information about the participants we selected.

8.3.2 Data Collection

To collect data through our controlled experiment, the first step consisted in (i) selecting the psychometric tests we have used to measure the attention- and

[4] We used the tool available at `https://www.statskingdom.com/sample_size_regression.html`

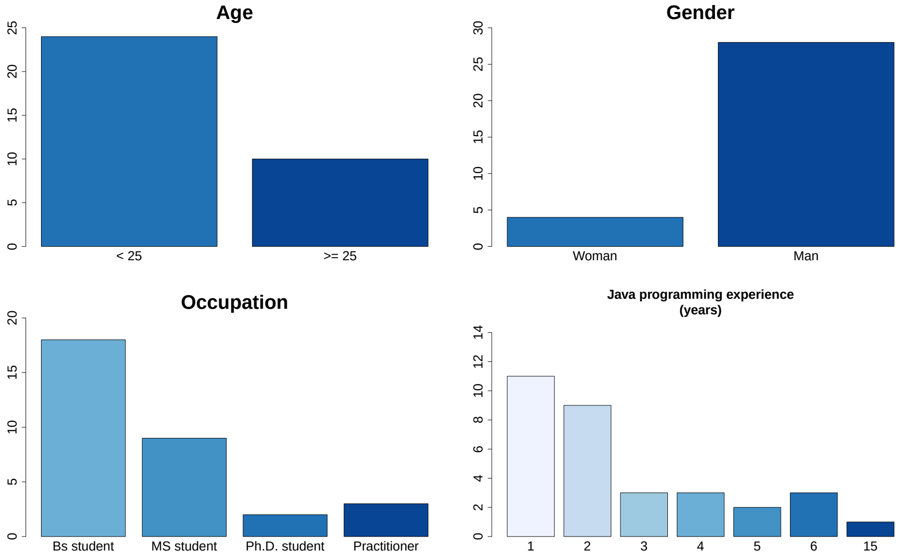


Figure 8.2: Demographic information about the participants.

memory-related factors we were interested in, and (ii) implementing them in a web-app that allowed us to administer such tests. The second step was to run the experiment. We describe below in detail these two steps.

Psychometric Tests.

We create a web-app that contains all psychometric tests. Through this web-app, participants could perform psychometric tests in any place and we could obtain automatically results of psychometric tests. To measure the attention-related factors, we used the Flanker Inhibitory Control and Attention Test [89]. In such a test, the participant is presented with five arrows, each one pointing either to the left or to the right. The goal of the participant is to indicate the direction of the central arrow as quickly as possible. Such a procedure is repeated several times and the arrow sets are presented in different ways. Some of them are preceded by a cue, *i.e.,* a hint about the time (*double cue*) and/or the place (*spatial cue*) in which the arrows will appear in the next few instants. Some others, instead, are presented without any cue (*i.e.,* they are just shown on the screen). When a cue is given, sometimes it is coherent with the time/location in which the arrows will appear, sometimes, instead, it is conflicting (*e.g.,* the cue is shown at the top, but the arrows appear at the bottom). It is possible to watch the video of the test in our replication package [39]. The web-app measures the response time (*RT*), *i.e.,* the time the participant takes to select answer. Besides, the web-app annotates, for each *RT*, the type of cue and if it was conflicting or not. In total, we aimed at collecting 128 *RT* measures for each participant; the test lasted about 10 minutes. After the first half (64 evaluations), participants could pause for as long as they wanted before continuing with the second half. We used the 128 *RT* measurements to compute the *alerting*, *orienting*, and *executive control* efficiency metrics through the following formulas:

- $\textit{alerting} = \text{mean}(RT_{\text{no cue}}) - \text{mean}(RT_{\text{double cue}})$
- $\textit{orienting} = \text{mean}(RT_{\text{central cue}}) - \text{mean}(RT_{\text{spatial cue}})$
- $\textit{executive control} = \text{mean}(RT_{\text{coherent}}) - \text{mean}(RT_{\text{conflicting}})$

The values of the metrics provide the efficiency of the *alerting*, *orienting*, and *executive control* networks of the participants, respectively. Higher values generally indicate that a given type of cue is effective and, therefore, the network (*i.e.,* alerting) is more efficient. For example, for *orienting*, a value greater than 0 indicates that the spatial cue allows the participants have better response times, *i.e.,* they are able to focus on the cued area. The range of response times is between 0 and 1700 ms.

The main test is preceded by a tutorial version of the test that lasts ∼3 minutes, in which we allow the participants to familiarize themselves with the web-app.

To measure the memory-related factors, we use two tests: the BTACT Word List Recall test [226] and the Symbol Digit Modalities Test [90]. The *BTACT Word List Recall* is part of the BTACT battery of cognitive processing tasks for adults. It allows to measure the *immediate recall*, or, more precisely, the immediate episodic memory for verbal material. The participants were asked to carefully listen to a set of 15 registered words, that we call C. Then, they were asked to repeat all the words they could remember in 90 seconds. Participants could use a button to indicate that they could not remember other words to proceed with the test. To assign a score, one of the authors listened to each recording offline, and manually annotated the words pronounced by each participant. Since most of the participants were non-native English speakers, we were tolerant for imperfect pronunciations, as long as it was clear that they referred to a word in the list C. This is the main reason why we did not use automated speech recognition.

For each participant, given the list of words pronounced, P, we compute the *immediate recall* as $|P \cap C|$, *i.e.,* the number of words correctly recalled. The *Symbol Digit Modalities Test* allows us to measure processing speed and working memory. In this test, participants were presented with a coding key mapping nine abstract symbols to numbers from 1 to 9. Participants memorize the mapping for the time they need. Then, they were presented 144 symbols in a random order with the mapping still being visible. Finally, the participants had 120 seconds to "decode", *i.e.,* to write in numbers, as many symbols as possible in the exact order they were presented. Participants could not skip symbols. We compute the *working memory* score as the total number of correctly decoded symbols.

Controlled Experiment Protocol. Before starting the experiment, we asked each participant to fill in a form through which they provided basic demographic information, *i.e.,* gender, education level, occupation, and years of experience, both with the Java programming language and overall. The experiment was divided, for each participant, in two sessions, held on different days. We divided the experiment in two days taking some risks and benefits. We risked that developers could refuse the participation to the experiment. Another risk is that participants could refuse to not continue in the second day. From the positive side, participants could not get too tired because we decrease the number of tasks for day. The alternative was to maintain the entire experiment on a single day. Using this alternative, we risked that developers refused the invitation to the experiment for the experiment lenght on a single day. Another risk with this alternative is that we obtained wrong results because participants was tired. Each session had the same structure. During each session, the first step was to administer the three previously described psychometric tests in the following order: BTACT Word List Recall (for *immediate recall*), Symbol Digit Modalities (for *working memory*), and, finally, Flanker Inhibitory Control and Attention Test (for *alerting*, *orienting*, and *executive control*). We measured memory-related factors at the beginning of the experiment because there could be the risk that participants was tired after to have performed the Flanker Inhibitory Control and Attention. These test have to be administered immediately before software development tasks because attention- and memory-related factors have a short-lived validity, as demonstrated in several previous studies [187, 124, 72]. The second step consisted in asking the participants to complete two of the four programming tasks outlined previously. In this phase, the author of the study, who supervise the experiment, shared a Java file containing (i) the description of the task (*i.e.,* the problem description from LeetCode) and (ii) either a partial solution (for bug fixing tasks) or the boilerplate for implementing the solution, such as the definition of the class and the method that LeetCode expected (for feature implementation tasks). Each participant had 30 minutes to complete each task. After 30 minutes, the author, who supervise the experiment, asked the participants to submit remotely the solution as it was.

To avoid biases due to the task execution order, we divided the participants in four groups. Depending on the group, a given participant was assigned the tasks in a different order and in different sessions. We defined the groups so as to assign at least an easy and a hard task, as well as a *bug fixing* and a *feature implementation* task for each session. This allowed us to control for fatigue (*e.g.,* the second task of the day could be performed systematically worse than the first one because the participant was tired) and learning (*e.g.,* participants could get quicker at completing coding tasks in the second session because they trained in the first one). We report the order in which the tasks were assigned to each group in Table 8.2. Because of the COVID-19 pandemic, it was not possible to completely control for the environment in which the tasks were performed (*e.g.,* run the study in a laboratory with the same equipment). As such, we performed the study adopting a remote setting, trying to recreate the lab setting we originally designed for the lab study. Specifically, each execution session was remotely supervised by the first author and, at each time, no more than two participants worked at the same time. Furthermore, the author of the study who controlled the experiment asked participants to turn on the microphone and the webcam, and to share the screen, so that it was possible to verify the presence of any distractions. The author who attended the experiment both guided the participants through the psychometric tests and the tasks, and ensured that the tasks were performed as they were intended (*e.g.,* force to submit the solution in the determined time frame, control if there were external distractions, such as phone calls). The psychometric tests were always administered to one participant at a time, to avoid that participants could affect each-other's behavior (*e.g.,* hearing the words in the Symbol Digit Modalities Test). The participants were not aware of the fact that the tasks were created using LeetCode problems. The author who attended the tasks made sure that they did not look for/use solutions found on the web. Participants shared their screens during the whole experimental sessions.

We operationalize the participants' performance in terms of *time* required to complete the task and *quality of the solution* (*correctness* and *readability*). We use them as dependent variables in our analysis in order to answer our research questions. To measure *time*, the author who supervise the experiment manually recorded the time at which each participant started each task and the time at which they reported that they concluded the task. We measure the time in minutes.

To measure the *correctness*, we relied on the test suite provided by LeetCode, as shown in Table 8.1. One of the authors copied and ran each solution given by the participants in LeetCode and measured the number of passed test cases for the task x, T_x^+. For feature implementation tasks, we compute the *correctness* simply as:

$$Correctness_x^{FI} = \frac{T_x^+}{T_x}$$

where T_x is the total number of test cases run by LeetCode for x. For bug fixing tasks, instead, we could not use the same formula: The two bug fixing tasks already came with a wrong solution. If developers left the provided solutions, without modifying them they would have achieved higher *correctness* on the task with the less buggy solution (the two buggy solutions had a different starting *correctness*). To avoid this, we measured, instead, the relative change in correctness: Ideally, a participant should achieve 100% *correctness* if he/she achieves 100% of passed tests, 0% if the *correctness* does not change compared to the initial solution, and

-100% if no tests pass. To achieve this, we used the following formula:

$$Correctness_x^{BF} = \begin{cases} \frac{T_x^+ - BT_x^+}{1 - BT_x^+}, & \text{if } T_x^+ \geq BT_x^+ \\ \frac{T_x^+ - BT_x^+}{BT_x^+}, & \text{otherwise} \end{cases}$$

where BT_x^+ is the number of tests passed with the buggy solution provided by the experimenters.

To measure *readability*, we used the approach defined by Scalabrino *et al.* [199]. While the model returns a binary assessment (*i.e., readable* or *unreadable*), it also provides a number which ranges between 0 and 1, which represents the estimated probability that the given snippet is readable. We use such a continuous value in this study. In this case, however, we only consider feature implementation tasks beacause the bug fixing is a task different from the writing of the code. We asked the participants to fix the bug and not to write readable code. The readability of the solutions of bug-fixing tasks, indeed, might strongly depend on the partial solution we provided and we did not explicitly ask developers to improve the readability of the provided solution. While we did not ask developers to write readable code in feature implementation tasks as well, it is worth noting that writing code from scratch forces developers to make choices that affect readability (*e.g.,* deciding identifiers' names).

Before running the experiment, we obtained the approval of the ethical board of our research institutions (ID number: ERB2021MCS5). Also, we ran a small pilot study with three additional participants (not involved in the main study), in order to test the web-app and the protocol and to spot any possible problem before starting the study. Participants of the pilot study declared that the study was feasible to performed it on two days. Thus, the protocol remained to be carried out on two days.

8.3.3 Data Analysis

To answer both our research questions, we initially compute the correlation between each continuous independent variable and the two dependent variables, to understand if there is any direct relationship between couples of dependent/independent variables. To do this, we use the Spearman rank correlation ρ [212].

Table 8.2: Task assignment for each group. 🐞 and 💧 indicate task types (*bug fixing* and *feature implementation*), while ︾ and ︽ indicate task difficulty (*easy* and *hard*).

Group	Session 1		Session 2	
	1st Task	2nd Task	1st Task	2nd Task
1	🐞︾	💧︽	💧︾	🐞︽
2	💧︾	🐞︽	🐞︾	💧︽
3	🐞︽	💧︾	💧︽	🐞︾
4	💧︽	🐞︾	🐞︽	💧︾

For the Spearman rank correlation, the correlation is **weak** if the coefficient is between -0.3 and +0.3, **moderate** if the coefficient is between -0.3 and -0.7 or between +0.3 and +0.7, **strong** if the coefficient is less than -0.7 or greater than +0.7 [209].

Then, to also account for interactions between independent variables, we combine them using explanatory regression models. Specifically, we use generalized linear regression models with a Gaussian link function. The independent variables we use for such models are *alerting*, *orienting*, *executive control*, *working memory*, *immediate recall*, *Java programming experience*, *task difficulty*, and *task type*. To answer RQ_1, RQ_2, and RQ_3 we use *correctness*, *time*, and *readability* as the dependent variables, respectively. Therefore, we define three models, one for each dependent variable. Before performing the models, we delete outlier until the skewness of all indipendent variables is less than 0.5. For each model, we report its explanatory power (R^2 and R^2_m), the AIC, and the significance obtained for each independent variable (p-values), to understand to what extent they explain *time*, *correctness*, or *readability*. If one of the variables obtains a p-value lower than 0.05, we reject the null hypothesis that such a variable does not explain the dependent variable. Additionally, we use backward stepwise elimination to gradually remove independent variables that give a non-significant benefit to the model and obtain a minimal model for explaining both our dependent variables. To this aim, we start with a full model containing all the independent variables, M_0. Then, we progressively remove the independent variable which is less likely to have a relationship with the dependent variable (*i.e.,* the one with the highest p-value, even if the it suggested statistical significance), we define a new model, M_1, and we measure its AIC. We repeat this steps, until the AIC of the model M_{i+1} is lower than the one of the previous version, M_i; in that case, we keep M_i as the minimal model.

Finally, to corroborate our findings, for RQ_1 we check if there is any significant difference in the independent variables between tasks correctly completed (*i.e.,* 100% correctness) and tasks with at least one failing test case (*i.e.,* correctness lower than 100%). We carry out a similar analysis for RQ_2: In this case, we check the difference between tasks completed before the time was up (*i.e.,* in less than 30 minutes) and when the time was over (*i.e.,* exactly 30 minutes). For RQ_3, we check if there is any significant difference between readable and unreadable solutions, based on the binary classification provided by the readability model. In both the cases, we adopt two hypothesis tests, depending on the variable type: We use the Mann-Whitney U test [142] for continuous variables such as time and readability, while we use the Fisher exact test [92] for categorical variables such as correctness. In RQ_1, the null hypotheses are: *"There is no difference in the independent variable x between tasks correctly completed and tasks with at least a bug"*, while, in RQ_2, the null hypotheses are: *"There is no difference in the independent variable x between tasks completed before the time was up and tasks completed when the time was over".* For each family of hypotheses, we adjust the p-values for multiple comparisons using the Benjamini and Hochberg procedure [50]. We also report the effect size, using the Cliff's delta [71], to understand the magnitude of differences observed. Cliff's delta δ lays in the interval [-1, 1]: the effect size is **negligible** for $|\delta| < 0.148$, **small** for $0.148 \leq |\delta| < 0.33$, **medium** for $0.33 \leq |\delta| < 0.474$, and **large** for $|\delta| \geq 0.474$ [71]. If $\delta > 0$, it means that the first distribution is larger than the second one, while the opposite happens otherwise [71].

8.3.4 Replication Package

All the anonymized data acquired in the experiment are available in our replication package [39], which also includes the four tasks (*i.e.,* description and solution of tasks) and the script used for statistical analysis.

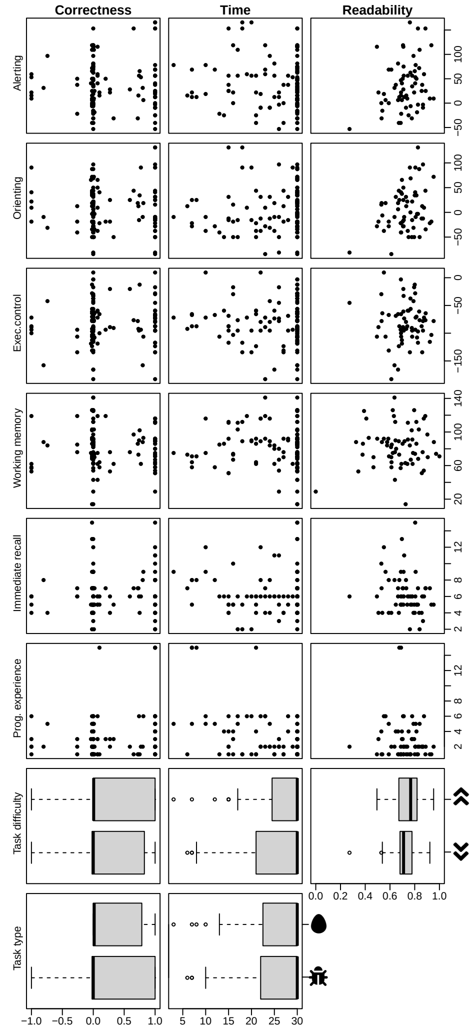


Figure 8.3: Relationships between the independent variables (y axis) and dependent variables (x axis). We use scatter plots for continuous variables and box plots for binary ones. 🐞 and ⬮ indicate task type (*bug fixing* and *feature implementation*), while ︾ and ︽ indicate the task difficulty (*easy* and *hard*).

8.4 Results

In this section, we provide empirical evidence to answer our research questions and discuss our results in Section 8.4.4. First of all, we want to verify if there are possible effects of fatigue and learning. To do this, we analyzed results of Day 1 and results of Day 2. For *correctness* and *time*, there is a data deterioration. Indeed, both for *correctness* and for *time*, we can see that in the second day there is a light worsening of results. Indeed, on Day 1, 69% of solutions are not completely corrected and 59% of solutions have been completed in 30 minutes. On Day 2, 79% of solutions are not completely corrected and 67% of solutions have been completed in 30 minutes. For *readability*, we obtain a stabilisation of results. Indeed, 45% of solutions was readable code for Day 1 and 46% of solution was readable for Day 2. These results demonstrate that the fatigue affects *correctness* and *time*, but developers write readable code in all cases. Overall, the participants achieved 26% of average *correctness* (21% for *bug fixing* and 32% for *feature implementation* tasks). 25% of the participants were able to achieve 100% correctness (30% for *bug fixing* and 22% for *feature implementation*). On average, the participants involved in our study completed the tasks in ∼25.5 minutes (25.4 for *bug fixing* and 25.6 for *feature implementation* tasks). 59% of the participants completed the task in 30 minutes, *i.e.,* they submitted when the time was over (58% for *bug fixing* and 61% for *feature implementation* tasks). Finally, 96.9% of the solutions wrote readable code in feature implementation tasks. Figure 8.3 plots the pairwise relationships between each independent variable and the

Table 8.3: Spearman rank correlations ρ between independent variables and dependent ones (significant correlations in bold).

	Full dataset		**FI dataset**
	Correctness	**Time**	**Readability**
Alerting	-0.006	-0.106	**0.234**
Orienting	-0.041	0.160	**0.208**
Executive control	0.076	-0.020	0.016
Working memory	0.009	-0.032	0.005
Immediate recall	0.086	-0.017	-0.189
Programming experience	0.164	**-0.324**	-0.073

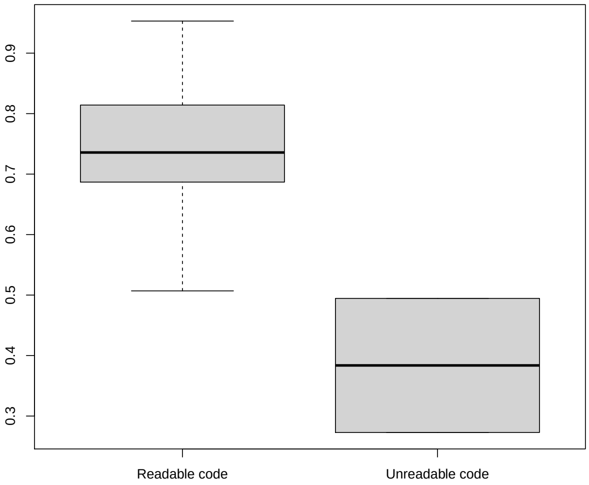


Figure 8.4: Boxplot of readability values related to feature implementation tasks.

three dependent ones we investigate, *i.e., correctness* (RQ_1), *time* (RQ_2), and *readability* (RQ_3). We use scatter plots for continuous variables and box plots for categorical (binary) ones. Before commenting this figure, as mentioned in Section 8.3, it is worth noting that some values of correctness could be between 0 and -1. Specifically, these values correspond to the correctness of bug fixing tasks. As described in Section 8.3, the correctness of bug fixing tasks could be equal to -100% when no test case passes on the final solution (*i.e.,* not even the ones that passed on the partial solution provided). It is also worth noting that for *readability* we do not report the relationship with the *task type* variable since we only considered feature implementation tasks, as explained in the design. The first insight we get from such a figure is that there is no clear relationship between pairs of independent and dependent variables. Interestingly, only a small difference in *correctness* and *time* can be noticed for task-related variables (*i.e., difficulty* and *type*). It is also possible to notice a slight correlation between readability and two attention-related factor, *i.e.,* orienting and alerting: higher attention seem to be associated to higher readability scores. This visual intuition is later quantitatively confirmed in the results of RQ_3.

Finally, we can state that *readability* is enough important to explain the outcome of a coding tasks. Differently, *correctness* and *time* play a vey marginal role in explaining the final outcome of a coding task.

8.4.1 RQ_1: Impact of Attention and Memory on Correctness

Table 8.3 reports the Spearman ρ correlation coefficients between each independent variable and *correctness*. The first clear fact that can be deduced from the correlations is that both attention-related factors (*alerting*, *orienting*, and *executive control*) and memory-related ones (*immediate recall* and *working memory*) achieve very low correlations with correctness. In particular, the highest correlation is observed for *immediate recall* (∼0.09). The *programming experience* achieve the highest correlation ($\rho \simeq 0.16$). No correlation, however, is significant, when the p-values are adjusted with the Benjamini and Hochberg procedure [50]. In absolute terms, all the correlations with single independent variables are very low.

We combine such variables using a generalized linear model, and we report in Table 8.4 (upper part), for each independent variable, the coefficient and the p-value obtained. The model confirms what the individual correlations suggested: The *programming experience* is the only important variable (p-value = 0.040). It is interesting to note that the *programming experience* in the full model is significant also putting it in relation with other factors. In addition, it is the only variable that remains in the minimal model after applying backward stepwise elimination. In such a model, it achieves, again, statistical significance (p-value = 0.027). The resulting model, however, has a very low explanatory power ($R^2_m = 0.03$ for the minimal model). Also in this case, no attention-related and memory-related factor is significant, and the only factor that remains after backward stepwise elimination is the *programming experience.*

Finally, Table 8.5 reports the results of the comparisons between distributions of independent variables in the two groups taken into account (*correct* and *incorrect*). While, again, no significant difference can be found, we observed a non-negligible (*small*) effect size for programming experience.

Table 8.4: Explanatory models for *correctness*, *time* and *readability*. For minimal models, we report the step of backward stepwise elimination at which each marginally relevant independent variable was removed. For each model, we also report AIC, R^2, and R^2_m.

Variable	**Correctness**			
	Full Model		**Minimal Model**	
	Coefficient	***p*-value**	**Coefficient**	***p*-value**
Intercept	-3.563e-02	0.882	0.15104	**0.028**
Alerting	8.347e-05	0.936	*Removed at step 2*	
Orienting	5.931e-04	0.598	*Removed at step 4*	
Executive control	1.332e-04	0.922	*Removed at step 3*	
Immediate recall	1.900e-02	0.389	*Removed at step 6*	
Working memory	-1.671e-04	0.939	*Removed at step 1*	
Prog. experience	3.707e-02	**0.040**	0.03816	**0.027**
Task type	1.183e-01	0.210	*Removed at step 7*	
Task difficulty	-5.998e-02	0.524	*Removed at step 5*	
	AIC: 212, $\mathbf{R^2}$: 0.06, $\mathbf{R^2_m}$: <0.01		**AIC**: 201, $\mathbf{R^2}$: 0.03, $\mathbf{R^2_m}$: 0.03	

Variable	**Time**			
	Full Model		**Minimal Model**	
	Coefficient	***p*-value**	**Coefficient**	***p*-value**
Intercept	29.155228	$<$ **0.001**	29.1137	$<$ **0.001**
Alerting	-0.022749	0.056	*Removed at step 7*	
Orienting	0.010888	0.398	*Removed at step 4*	
Executive control	0.005244	0.735	*Removed at step 3*	
Immediate recall	-0.073613	0.770	*Removed at step 2*	
Working memory	0.028154	0.260	*Removed at step 5*	
Prog. experience	-1.246597	$<$ **0.001**	-1.2407	$<$ **0.001**
Task type	-1.542033	0.153	*Removed at step 1*	
Task difficulty	0.154936	0.885	*Removed at step 6*	
	AIC: 836, $\mathbf{R^2}$: 0.28, $\mathbf{R^2_m}$: 0.23		**AIC**: 828, $\mathbf{R^2}$: 0.23, $\mathbf{R^2_m}$: 0.24	

Variable	**Readability**			
	Full Model		**Minimal Model**	
	Coefficient	***p*-value**	**Coefficient**	***p*-value**
Intercept	0.722830	**<0.001**	0.722350	$<$ **0.001**
Alerting	0.000439	0.182	*Removed at step 6*	
Orienting	0.000717	**0.048**	0.000876	**0.012**
Executive control	0.000226	0.597	*Removed at step 2*	
Immediate recall	-0.007807	0.264	*Removed at step 5*	
Working memory	0.000587	0.396	*Removed at step 4*	
Prog. experience	-0.002530	0.655	*Removed at step 1*	
Task type	//	//	//	//
Task difficulty	0.020942	0.486	*Removed at step 3*	
	AIC: -82, $\mathbf{R^2}$: 0.18, $\mathbf{R^2_m}$: 0.08		**AIC**: -88, $\mathbf{R^2}$: 0.10, $\mathbf{R^2_m}$: 0.08	

Summary of RQ_1. Attention- and memory-related factors do not correlate with the correctness of a task. Conversely, programming experience shows a statistically significant correlation with correctness, albeit with small effect size.

8.4.2 RQ_2: Impact of Attention and Memory on Time

As previously done for RQ_1, we compute the Spearman ρ correlation coefficients between our independent variables and *time*. We report the results in Table 8.3. We observe that, while the correlations are low, they are generally higher than the ones achieved for *correctness*. In this case, one of the attention-related variables, *i.e., orienting*, achieves a relatively higher correlation coefficient ($\rho \simeq 0.16$). Such a correlation, however, is not significant, as all the ones with all the other independent variables except for one, *i.e., programming experience*. Such a correlation is high compared to the others ($\rho \simeq 0.32$), but it is still weak in absolute terms. With this correlation, we can state that participants having more experience tend to work faster.

We report the generalized linear model for *time* in Table 8.4 (middle part). *Programming experience* appears to be, also for *time*, the most important factor (p-value < 0.001). It is worth noting that one of the attention-related factors, *i.e., alerting*, is almost significant (p-value $= 0.056$) in the full model. However, such a factor is not selected in the minimal model. *Programming experience* is confirmed to be, again, the only relevant factor. For *time*, the minimal model achieves a much higher explanatory power than the one we built for *correctness* ($R^2_m = 0.24$).

Finally, we report in Table 8.5 the results of the comparisons between distributions of independent variables in the groups *time-up* (task finished in exactly 30 minutes) and *non-time-up* (task finished before the time was over). In this case, we have two variables for which the difference is non-negligible (*small*) in terms of effect size, *i.e., orienting* and *programming experience*. The last one, in particular, is significantly higher for the coding tasks completed before the time was over, as expected.

Table 8.5: Comparisons between independent variables in groups (correct vs incorrect for RQ_1, time-up vs non-time-up for RQ_2, and readable vs unreadable for RQ_3) using Mann-Withney (†) and Fisher (‡) tests.

Comparison	**Variable**	***p*-value**	**Cliff's *d***
Correct *vs.* Incorrect	Alerting†	0.973	-0.039 (negl.)
	Orienting†	0.973	-0.038 (negl.)
	Executive control†	0.973	0.048 (negl.)
	Immediate recall†	0.973	0.077 (negl.)
	Working memory†	0.973	0.022 (negl.)
	Programming experience†	0.154	**0.265** (small)
	Task difficulty‡	1.000	0.020 (negl.)
	Task type‡	0.973	-0.102 (negl.)
Time-up *vs.* Non-time-up	Alerting†	0.746	-0.093 (negl.)
	Orienting†	0.381	**0.174** (small)
	Executive control†	0.932	0.009 (negl.)
	Immediate recall†	0.932	0.029 (negl.)
	Working memory†	0.920	**-0.059 (negl.)**
	Programming experience†	**0.010**	**-0.323** (small)
	Task difficulty‡	0.746	0.097 (negl.)
	Task type‡	0.932	0.032 (negl.)
Readable *vs.* Unreadable	Alerting†	1.000	0.089 (negl.)
	Orienting†	0.447	**0.782** (large)
	Executive control†	1.000	-0.056 (negl.)
	Immediate recall†	1.000	-0.064 (negl.)
	Working memory†	0.912	**-0.379** (medi.)
	Programming experience†	0.912	**-0.355** (medi.)
	Task difficulty‡	1.000	0.016 (negl.)

Summary of RQ_2. *Alerting* and *orienting* are attention-related metrics that show some signs of (non-significant) correlation with the time needed to complete a task. Similarly to what observed for correctness, the programming experience shows the highest correlation with time.

8.4.3 RQ_3: Impact of Attention and Memory on Code Readability

As discussed in RQ_1 and RQ_2, in Table 8.3 we report the Spearman ρ correlation coefficients between our independent variables and *readability.* Differently from the previous research questions, we can observe small correlations between readability and attention-related factors. Specifically, there is correlation with *alerting* (ρ = 0.234) and *orienting* (ρ = 0.208). In addition, there are negative very small correlations on *immediate recall* (ρ = -0.189) and *programming experience* (ρ = -0.073). When the p-values are adjusted with the Benjamini and Hochberg procedure [50], however, we observe no significant correlation.

We report the generalized linear model obtained for *readability* in Table 8.4 (lower part). In this case, *programming experience* is not an important factor because the p-value is not significant (0.655). In the full model, we can see that the only significant factor is *orienting* (p-value = 0.048). This factor is important also in the minimal model (p-value = 0.012).

At the end, we report in Table 8.5 the results of the comparisons between distributions of independent variables in the groups *readable* and *unreadable* solutions. In terms of effect size, we have two variables for which the difference is non-negligible: *working memory* and *programming experience* (*medium*), and *orienting* (*large*). The adjusted p-values, however, show no significant difference. This is most likely due to the lower number of data points considered with respect to the two other dependent variable, given that we only considered feature implementation tasks for *readability.*

Summary of RQ_3. *Orienting* is significantly associated with the code readability in the generalized linear model, and orienting of developers who produce readable code is largely higher compared to the ones who produce unreadable code. However, such a difference is not significant, mostly due to the lower number of data-points considered for this RQ.

8.4.4 Discussion

In our study, we aimed at assessing to what extent cognitive aspects are correlated with developers' performance. In particular, we investigated to what extent attention and memory correlate with the time required to complete a development task and the quality of the solution, both in terms of correctness and readability. The empirical evidence provided by our study suggests a negative result for memory and attention. Indeed, neither attention nor memory cannot explain how developers complete coding tasks in terms of correctness and time. While we obtain a negative result for time and correctness, we obtain a positive result for readability. This result enforces other similar previous studies [173, 207] that showed that the attention is an important factor for program comprehension. Therefore, we can conclude that developers with high attention not only understand code better, but also write more readable code. On the other hand, we did not observe a correlation between readability and memory-related factors. It is worth noting that explaining the outcome of the activity of code writing, either for implementing new features or for fixing bugs, is an ambitious task, as the low explanatory power obtained through by our models for *time* and, above all, *correctness* show. This means that, probably, many other factors (including other cognitive ones) should be taken into account to achieve results usable in practice.

As for correctness and time, our results show that there is a single variable that clearly outperforms all the others: the *programming experience* in the specific programming language (Java, in our study). Therefore, we can say, from our results, that, while there are some weak relations between cognitive aspects and the two variables we considered, *correctness* and *time*, experience alone is a far better variable to explain both of them. In other words, there appears to be no “shortcut” for improving in coding tasks from the point of view of cognitive human aspects. However, the association observed between *readability* and *attention* suggests that it should be possible to devise a training that allow developers to improve cognitive functions [161] (attention, in our case). Such an improvement could be beneficial for writing readable code.

Another interesting phenomenon we observed is the following: In general, individual differences (in this case, in terms of programming experience) allow to better explain the outcome than the differences among the tasks (in our study, type and difficulty).

This means that the skills of a developer are far more important than the characteristics of the task at hand for determining the success in completing it and the time required. Also if we analyze only two type of tasks, this is in line with what was previously observed for code understandability [198]. We can infer that developers with low experience that work on a task might take more time and might produce worse results *regardless of the task at hand.* This allows us to give a clear recommendation to practitioners, which is already applied in many contexts because of anecdotical evidence: *Less experienced developers should benefit from more experienced developers to achieve higher quality in the final software product.*

8.5 Threats to Validity

As it is the case for any empirical study validity of our conclusions might have been threatened in several ways.

Threat to Construct Validity. The biggest threats to internal validity are related to the methodology used to measure the independent variables, above all the attention- and memory-related ones. As described in Section 8.3, we use state-of-the-art psychometric tests to achieve this goal and we replicated them in our web-app. It might be possible that implementation errors have caused wrong measurements. We thoroughly tested the web-app to avoid this. In addition, there is the possibility that we use existing psychometric tests in a wrong way. Attention and memory could be measured with dedicated and specialized psychometric tests who are far from our knowledge. To the best of our knowledge, we could not find dedicated, specialized psychometric tests related to the programming. Thus, we used tests that have been already applied in software engineering [163]. The accuracy with which we could reliably measure reaction times in the Flanker

Inhibitory Control and Attention Test (for attention-related metrics) was within tenths of a second; state-of-the-art measurement provide the measure of reaction times with the precision of thousandths of a second. We believe that this is a negligible limitation: the mean reaction time we obtained is $\sim 681ms$, meaning that the maximum error due to the lack of precision (±50ms) would be at most $\sim 7.3\%$. Such a level of precision would be mostly irrelevant. The original version of the Flanker Inhibitory Control and Attention Test [89] we used for measuring attention-related factors provided three blocks with 96 evaluations each (288 total evaluations). We used reduced version of such a test [232], which provides two blocks of 64 evaluations each (128 total evaluations). This allowed us to reduce the effort for developers. It could be possible that psychometric tests might cause fatigue in the developer and, therefore, reduce their ability of correctly completing the tasks or increasing the time required to do so. To reduce this effect, we let developer rest for a few minutes after the tests, before starting the tasks. We could not avoid this, since measurements of attention-related factor may depend on the context in which they are taken [147]: Measuring the variables may provide irrelevant measures.

Threat to Internal Validity. One of threats to internal validity is related to the measurement of *programming experience.* We asked developers to indicate the number of years of experience in Java. Siegmund *et al.* [206] showed that, while this is the most commonly used approach, other self-estimation questions might be more relevant for students (*e.g.,* self-assessment of experience compared to other class mates). Therefore, alternative and more reliable measures of *programming experience* might allow to achieve even higher correlations with both *correctness* and *time.*

Another threat is related to the definition of the tasks. It could be argued that the bug-injection methodology we used is not completely realistic since we arbitrarily modified the code. However, a similar methodology has been adopted in previous studies as well [114, 238, 237].

Also, it could be argued that the description of the problems taken from LeetCode was not necessarily understandable to all developers. We assume that, being exposed every day to the many users of LeetCode, any understandability problem was removed from the description in time.

Another threat is related to the task choice. We chose tasks considering different difficulty levels: However, the perceived difficulty is subjective by nature. For example, the task *Find and Replace Pattern* could be easy for a developer that has experience with regular expressions, but it could be difficult for others.

There could be other factors, different from attention and associated with natural language competencies, which better explain code readability. For instance, proficiency in English writing, knowledge of the English vocabulary. These factors are not considered in this study. Hence, the recommendation to devise exercises to improve attention and have a positive effect on code readability might be overridden by the need to improve English language knowledge. We plan to execute an extension of this study, measuring the effect of natural language competencies of participants

Another threat is induced by the time-out: the time is not necessary to complete the entire software development task. We select this time also after the execution of pilot study. Participants of pilot study confirmed that the time was sufficient to complete software development tasks. For this reason and for decreasing possible fatiche and learning effects, we decide to adopt this time.

Threat to External Validity. Our findings may be mostly related to the specific sample of developers we involved in the study. We included a total of 32 developers from four different countries. This is sufficient to avoid biases related to common education background and occupation. Our a-priori power analysis suggested that a sample of 26 developers would have been sufficient to achieve 80% power. To further verify that our sample is not too small, we also run post-hoc power analysis for our two regression models. We achieve 83.1% power for the *correctness* model (RQ_1), 99.9% for the *time* model (RQ_2) and 95.0% for the *readability* model (RQ_3). Therefore, we can confidently conclude that the data on which we based our analyses are sufficient to draw the conclusions we made.

The only hints at the fact a bigger sample could have resulted in additional significant differences are given by (i) the non-negligible effect size we obtained when comparing the *orienting* values between the *time-up* and the *non-time-up* groups, and (ii) the almost significant p-value of *alerting* obtained in the full regression model that predicts *time* (p-value = 0.056). In the former, the p-value of the comparison (0.38) is quite far from the threshold we chose for significance

(0.05), and the effect size, while being *small* (0.174), is very close to the *negligible* threshold (0.148). As for the latter, instead, it can be noticed, given the coefficient assigned to the *alerting*, such a value has an influence on the time fitted by the model between -3.6 and +1.2 minutes. For comparison, the effect of *programming experience* is between -18.8 and -1.3 minutes. In other words, it would require developers to have a relatively high *alerting* efficiency for having a small benefit in terms of time, according to the model, regardless of the significance of the variable.

Another threat to external validity is the lack of control for the programming experience. As described in Section 8.3, we recruited 32 participants without considering their programming experience. As written above, we invited participants from four different countries. As a consequence, their background is heterogeneous. Unfortunately, given the lack of control for the programming experience, there is a developer with 15 years of Java experience that increase the average experience. However, if we removed the outlier, we do not obtain dissimilar results. We have not dissimilar results because for many solutions the correctness is equal to 0 and the time is equal to 30.

Finally, another threat is the fact that we expect a R^2 equal to 0.13. In literature there is the study of Rasch *et al.* [188] in which authors obtain a R^2 of 0.54. This study is different from our study beacause authors performed a questionnaire with an high number of participants. The experiment is structurally different from our controlled experiment.

8.6 Final Remarks

Attention and memory have been shown to be related to the outcome of different kinds of tasks (*e.g.,* driving or solving mathematical problems), and they have also been used in the context of Software Engineering and Software Security (*e.g.,* for determining the usability of security APIs [163]). For this reason, we theorize that attention- and memory-related factors also play a role in coding tasks, which are at the base of software development and evolution. *The goal of this study is understand if cognitive human aspects can influence the resolution of coding tasks.* We conducted a controlled experiment with 32 developers, and we asked them to complete 4 tasks each. We measured three attention-related metrics and two memory-related ones, widely used in the literature. We aim at predicting the *time* needed to complete the tasks and the quality of the solutions, both in terms of *correctness* and *readability*. We check the relationship between independent variables (including also *programming experience*, *task difficulty*, and *task type*) and the three dependent ones both by using single correlations and by combining them in a regression model. On the one hand, we obtained a *negative result*: neither attention- nor memory-related factor play a role in explaining *correctness* and *time*. Programming experience is the only significant factor, much more important than the task difficulty and the task type. On the other hand, we observed that an attention-related factor (orienting) is associated with higher capability of writing readable solutions.

From our results, we can derive some lessons learned:

- developers should train their attention level to write readable code;
- experienced developers should always support the less experienced ones when the goal is to achieve higher quality in the final software product;
- researchers should study what personal characteristics can help developers to implement quality code. In this way, they could propose to developers specific training sessions for their development tasks.

Replications of our study are needed to further corroborate our findings. Also, future research could focus on other cognitive aspects, such as *intelligence* (*e.g.,* through IQ).

CHAPTER 9

Conclusion

Software developers need to read and comprehend the code before making any change on it. Thus, code reading is one of the principal activities performed by developers. For this reason, researchers have identified several factors that influence code readability [146, 164, 48] and—on the basis of these factors—they have proposed different models to automatically assess code readability [59, 181, 80, 200, 199].

In a preliminary study conducted to understand what are real motivations that push developers to improve the quality of source code through refactoring operations, we observe that very often refactoring operations have been performed to improve code readability. Such a result motivated this thesis: Studying code readability during the evolution of a complex software system. Indeed, previous studies on code readability have been conducted on single static code snippets without considering the natural and continuous evolution of a software system.

In the context of this thesis, we analyzed three different aspects related to code readability: (i) the importance of code readability for developers; (ii) how readability evolves in software systems; (iii) if cognitive human aspects can influence code readability.

Regarding the *first aspect*, we obtained that 83.8% of developers consider code readability as an important factor in their source code writing activities. As for the *second aspect*, our results showed that code readability of a file can undergo small changes during the development of a software system: A file is created readable and it remains readable or vice versa. Finally, turning to the *third aspect*, we obtained that the orienting network, one of the attention-related factors, correlates positively with code readability.

The results achieved in the context of this thesis can be used to derive the following lessons learned that could be useful to identify new research directions:

- *Identifying "when" to trigger refactoring recommendations.* Product- and process-related factors, including *code readability*, contribute to trigger refactoring operations. Such empirical evidence represents the basis for future approaches able to predict "when", during a project's evolution, refactoring recommendations are triggered. Indeed, such an aspect is currently ignored in the refactoring recommenders literature, with researchers focusing their attention on the core problem of generating meaningful recommendations. However, recommending refactorings in a "context" in which developers do not feel the need to refactor their code is unlikely to provide benefits. An interesting research direction in this field is to build models able to predict when refactoring recommendations would be welcome by software developers.

- *On learning refactoring operations to improve automatically code readability.* To improve code readability, one possibility worth exploiting in the future is the application of deep learning techniques to refactoring recommenders. Indeed, recent work has already shown the possibility to learn from code changes [225]. However, no previous work has attempted to design refactoring recommenders learning from developers' activities what meaningful refactoring is in a given context.

- *Lack of production-ready tools.* The research community has developed many good approaches for supporting specific refactorings and these approaches find little application in practice. As for *code readability*, the rename refactoring operations are suggested by reviewers to expand abbreviations used in the identifiers of the code contributed in PRs. These recommendations could be easily generated at commit time by one of the many approaches proposed in the literature for the automatic expansion of abbreviations (see *e.g.,* the study by Lawrie *et al.* [128]). However, the lack of production-ready tools could be the reason for the lack of adoption of such techniques. This is an opportunity not only for researchers, but also for developers interested in building tools useful for the partial automation of code review activities.

- *New readability prediction models specifically designed to classify changes instead of single snapshots would be necessary.* In the research community, there are different studies on readability prediction models, but these models operate on single snapshots. With the results achieved in our mining study on the evolution of code readability (Chapter 7), there is the possibility to define an accurate readability evolution model using a discrete-time Markov chain.

- *Creation of specific cognitive training sessions for developers.* Our results of the empirical study in Chapter 8 indicate that there is a correlation between personal characteristics (*i.e.,* cognitive human factors) and code readability. On the basis of these results, researchers could create specific cognitive training sessions for developers [161]. Such cognitive training sessions can help developers in the improvement of their personal characteristics useful for development activities, in general, and for code readability, in particular.

Acknowledgements

When I enrolled at the University of Molise in 2012, I never thought I would live these wonderful years of doctoral studies. Despite the many difficulties I encountered along the way and the historical period in which we lived, these 3 years of doctorate have served me to grow professionally and personally. I think that if there were not all these difficulties, perhaps I would never have achieved the results obtained in this thesis. In this small University I have lived many small things, of which I have many memories and many emotions that I carry with me. Without them, I would never have become the woman I am now. This long journey has given me the opportunity to meet some great people and I do not hide that my life without them would not be the same.

I had the possibility to meet the best advisor I could ask for, Prof. Rocco Oliveto. First of all, I have to thank him because he has allowed me to have my current preparation. He is not only a Professor, but he is also a friend, a brother, who is always ready to welcome you not as a member of his laboratory but as a member of his family. Yes, because our laboratory is not only a place of work, but also a small family.

In these last 3 years, I had the opportunity to have fantastic colleagues at my side: Simone, Gennaro, Giovanni, Michele and Emanuela. There is no order of importance for me, because for me they are all at the same level. I thank them for enduring my fears, my doubts and my anxieties. I thank them for rejoicing with me and for reassuring me when the need arose. Thank you for making our laboratory unique!

Thanks to my best friend Vincenzo for being there during these 10 years, for being my sidekick and my first fan. I thank him for always being there, if only to tell me: "Don't worry!"

Thanks to my best friends Rosangela and Federica for reminding me that friendship is important and should never be put aside.

Thanks to my parents for supporting me in every single choice I made, for listening to all my stories, for having suffered and rejoiced with me.

These 10 years have allowed me to meet also my boyfriend, Franz. In recent months, he has been instrumental in helping me complete my journey. He supported me in every single decision I made, and he put up with all of my moments of freaking out. He is my hearth.

A special thanks also goes to my other lab colleagues, Andrea, Davide and Umberto, and to all the guys who in these years have been part of our laboratory (who reads knows who I'm talking about).

The doctorate gives you the opportunity to expand your knowledge. Despite the pandemic, I had the opportunity to meet people who contributed to my professional growth: prof. Alexander Serebrenik, prof. Gabriele Bavota, prof. Massimiliano Di Penta, and prof.ssa Nicole Novielli. I am honored to have worked with you. Thank you!

Valentina Piantadosi
Isernia, May 6th, 2022

Appendices

APPENDIX A

Publications

J1 J. Pantiuchina, F. Zampetti, S. Scalabrino, **V. Piantadosi**, R. Oliveto, G. Bavota, M. Di Penta. *Why Developers Refactor Source Code: A Mining-based Study.* Transactions on Software Engineering and Methodology, 2020. DOI 10.1145/3408302.

J2 **V. Piantadosi**, F. Fierro, S. Scalabrino, A. Serebrenik, R. Oliveto. *How does code readability change during software Evolution.* Empirical Software Engineering. DOI 10.1007/s10664-020-09886-9.

J3 **V. Piantadosi**, S. Scalabrino, A. Serebrenik, N. Novielli, R. Oliveto *Do Attention and Memory Explain the Performance of Software Developers?.* Submitted at Empirical Software Engineering, 2022.

A.1 Other Publications

C1 **V.Piantadosi**, S. Scalabrino, R. Oliveto. *Fixing of security vulnerabilities in open source projects: A case study of Apache HTTP Server and Apache Tomcat.* IEEE Conference on Software Testing, Validation and Verification (ICST), 2019. DOI 10.1109/ICST.2019.00017.

J4 S. Scalabrino, G. Bavota, M. Linares-Vásquez, **V. Piantadosi**, M. Lanza, R. Oliveto. *API compatibility issues in Android: Causes and effectiveness of data-driven detection techniques.* Empirical Software Engineering, 2020. DOI 10.1007/s10664-020-09877-w.

J5 **V. Piantadosi**, G. Rosa, D. Placella, S. Scalabrino, R. Oliveto. *Detecting Functional and Security-Related Issues in Smart Contracts: A Systematic Literature Review.* Submitted at Software Practice and Experience, 2021.

Bibliography

[1] 52north/wps pull request #73. https://github.com/52North/WPS/pull/73.

[2] apache/fineract pull request #366. https://github.com/apache/fineract/pull/366.

[3] cbeust/testng pull request #1481. https://github.com/cbeust/testng/pull/1481.

[4] Code metrics for java code by means of static analysis.

[5] confluentinc/kafka-connect-elasticsearch pull request #251. https://github.com/confluentinc/kafka-connect-elasticsearch/pull/251.

[6] dropwizard/dropwizard pull request #488. https://github.com/dropwizard/dropwizard/pull/488.

[7] Dspace/dspace pull request #1083. https://github.com/DSpace/DSpace/pull/1083.

[8] Dspace/dspace pull request #324. https://github.com/DSpace/DSpace/pull/324.

[9] eclipse/microprofile-fault-tolerance pull request #363. https://github.com/eclipse/microprofile-fault-tolerance/pull/363.

[10] google/error-prone pull request #1071. https://github.com/google/error-prone/pull/1071.

[11] kiegroup/optaplanner pull request #150. https://github.com/kiegroup/optaplanner/pull/150.

[12] Leetcode problem: Duplicate zeros.

[13] Leetcode problem: Find and replace pattern.

[14] Leetcode problem: Regular expression matching.

[15] Leetcode problem: Sort the matrix diagonally.

[16] minio/minio-java pull request #238. https://github.com/minio/minio-java/pull/238.

[17] Pmd source code analyzer.

[18] samtools/htsjdk pull request #1067. https://github.com/samtools/htsjdk/pull/1067.

[19] samtools/htsjdk pull request #599. https://github.com/samtools/htsjdk/pull/599.

[20] spring-io/sagan pull request #179. https://github.com/spring-io/sagan/issues/179.

[21] spring-io/sagan pull request #317. https://github.com/spring-io/sagan/pull/317.

[22] spring-projects/spring-amqp pull request #346. https://github.com/spring-projects/spring-amqp/pull/346.

[23] Teamamaze/amazefilemanager pull request #1577. https://github.com/TeamAmaze/AmazeFileManager/pull/1577.

[24] zalando/nakadi pull request #626. https://github.com/zalando/nakadi/pull/626.

[25] zalando/nakadi/ pull request #933. https://github.com/zalando/nakadi/pull/933.

[26] zhcet-amu/zhcet-web pull request #136. https://github.com/zhcet-amu/zhcet-web/pull/136.

[27] H. Akaike. Information theory and an extension of the maximum likelihood principle. In *2nd International Symposium on Information Theory*, 1973.

[28] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Learning natural coding conventions. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 281–293, 2014.

[29] J. Allan. *Cognitions*, pages 441–441. Springer New York, New York, NY, 2013.

[30] A. Almogahed, M. Omar, and N. H. Zakaria. Categorization refactoring techniques based on their effect on software quality attributes. *International Journal of Innovative Techno, logy and Exploring Engineering (IJITEE)*, pages 439–445. 2019.

[31] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini. On the impact of refactoring on the relationship between quality attributes and design metrics. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–11. IEEE, 2019.

[32] U. Alon, M. Zilberstein, O. Levy, and E. Yahav. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29. Jan. 2019.

[33] M. Alshayeb. Empirical investigation of refactoring effect on software quality. *Information and Software Technology*, 51(9):1319 – 1326. 2009.

[34] Anonymous. Replication Package. `https://github.com/replication-package/why-refactoring`, 2018.

[35] N. Anquetil and T. Lethbridge. Experiments with clustering as a software remodularization method. In *WCRE*, pages 235–255, 1999.

[36] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 2008 conference of the Centre for Advanced Studies on Collaborative Research*, page 23. IBM, 2008.

[37] D. Arcelli, V. Cortellessa, and C. Trubiani. Antipattern-based model refactoring for software performance improvement. In *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*, pages 33–42, 2012.

[38] V. Arnaoudova, M. Di Penta, G. Antoniol, and Y.-G. Guéhéneuc. A new family of software anti-patterns: Linguistic anti-patterns. In *17th European Conference on Software Maintenance and Reengineering, CSMR 2013*, pages 187–196, 2013.

[39] A. Authors. Replication package of "do attention and memory explain the performance of software developers?", ????

[40] A. D. Baddeley. Working memory. *Philosophical Transactions of the Royal Society of London. B, Biological Sciences*, 302(1110):311–324. The Royal Society London, 1983.

[41] D. Bates, M. Mächler, B. Bolker, and S. Walker. Fitting linear mixed-effects models using lme4. *Journal of Statistical Software*, 67(1):1–48. 2015.

[42] G. Bavota. Using structural and semantic information to support software refactoring. In *34th International Conference on Software Engineering, ICSE 2012*, pages 1479–1482. IEEE Computer Society, 2012.

[43] G. Bavota, B. D. Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo. When does a refactoring induce bugs? an empirical study. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 104–113, 2012.

[44] G. Bavota, A. De Lucia, M. Di Penta, R. Oliveto, and F. Palomba. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software*, 107:1–14. Elsevier, 2015.

[45] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Automating extract class refactoring: an improved method and its evaluation. *Empirical Software Engineering*, pages 1–48. 2013.

[46] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Recommending refactoring operations in large software systems. In *Recommendation Systems in Software Engineering*, pages 387–419. Springer Berlin Heidelberg, 2014.

[47] G. Bavota, A. D. Lucia, A. Marcus, and R. Oliveto. Using structural and semantic measures to improve software modularization. *Empirical Software Engineering*, 18(5):901–932. 2013.

[48] K. Beck. *Implementation Patterns.* Addison Wesley, 2007.

[49] J. G. Benjafield, D. Smilek, and A. Kingstone. *Cognition (4th ed.).* New York: Oxford University Press, 2010.

[50] Y. Benjamini and Y. Hochberg. Controlling the false discovery rate: a practical and powerful approach to multiple testing. *Journal of the Royal statistical society: series B (Methodological)*, 57(1):289–300. Wiley Online Library, 1995.

[51] K. H. Bennett and V. T. Rajlich. Software maintenance and evolution: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, 2000.

[52] E. Bialystok and A.-M. DePape. Musical expertise, bilingualism, and executive functioning. *Journal of Experimental Psychology: Human Perception and Performance*, 35(2):565. American Psychological Association, 2009.

[53] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2009, Amsterdam, The Netherlands, August 24-28, 2009*, pages 121–130. ACM, 2009.

[54] B. W. Boehm, J. R. Brown, H. Kaspar, M. Lipow, G. J. Macleod, and M. J. Merrit. *Characteristics of Software Quality*, volume 1 of *TRW series of software technology.* Elsevier, 1978.

[55] B. W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, R. Madachy, and B. Steece. *Software Cost Estimation with Cocomo II with Cdrom.* Prentice Hall PTR, USA, 1st edition, 2000.

[56] W. J. Brown, R. C. Malveau, W. H. Brown, H. W. McCormick III, and T. J. Mowbray. *Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis.* John Wiley and Sons, 1^{st} edition, Mar. 1998.

[57] Y. Brun, T. Lin, J. E. Somerville, E. Myers, and N. C. Ebner. Blindspots in python and java apis result in vulnerable code. *arXiv preprint arXiv:2103.06091.* 2021.

[58] R. P. Buse and W. R. Weimer. A metric for software readability. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 121–130. ACM, 2008.

[59] R. P. Buse and W. R. Weimer. Learning a metric for code readability. *Software Engineering, IEEE Transactions on*, 36(4):546–558. IEEE, 2010.

[60] J. Businge, A. Serebrenik, and M. G. J. van den Brand. Eclipse API usage: the good and the bad. *Softw. Qual. J.*, 23(1):107–141. 2015.

[61] G. Canfora, L. Cerulo, M. Cimitile, and M. D. Penta. How changes affect software entropy: an empirical study. *Empirical Software Engineering*, 19(1):1–38. 2014.

[62] A. Capiluppi, M. Morisio, and P. Lago. Evolution of understandability in oss projects. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.*, pages 58–66. IEEE, 2004.

[63] J. Cappos, Y. Zhuang, D. Oliveira, M. Rosenthal, and K.-C. Yeh. Vulnerabilities as blind spots in developer's heuristic-based decision-making processes. In *Proceedings of the 2014 New Security Paradigms Workshop*, pages 53–62, 2014.

[64] D. Cedrim, A. Garcia, M. Mongiovi, R. Gheyi, L. Sousa, R. de Mello, B. Fonseca, M. Ribeiro, and A. Chávez. Understanding the impact of refactoring on smells: A longitudinal study of 23 software projects. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 465–475, 2017.

[65] A. Chatzigeorgiou and A. Manakos. Investigating the evolution of code smells in object-oriented systems. *Innovations in Systems and Software Engineering*, 10(1):3–18. Springer, 2014.

[66] A. Chávez, I. Ferreira, E. Fernandes, D. Cedrim, and A. Garcia. How does refactoring affect internal quality attributes?: A multi-project study. In *Proceedings of the 31st Brazilian Symposium on Software Engineering*, pages 74–83, 2017.

[67] C. Chen, R. Alfayez, K. Srisopha, L. Shi, and B. Boehm. Evaluating human-assessed software maintainability metrics. In *National Software Application Conference*, pages 120–132. Springer, 2016.

[68] C.-T. Chen, Y. C. Cheng, C.-Y. Hsieh, and I.-L. Wu. Exception handling refactorings: Directed by goals and driven by bug fixing. *Journal of Systems and Software*, 82(2):333 – 345. 2009.

[69] M. R. Chernick. *Bootstrap methods: A guide for practitioners and researchers*, volume 619. John Wiley & Sons, 2011.

[70] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering (TSE)*, 20(6):476–493. June 1994.

[71] N. Cliff. Dominance statistics: Ordinal analyses to answer ordinal questions. *Psychological bulletin*, 114(3):494. American Psychological Association, 1993.

[72] A. Conway, C. Jarrold, and A. Miyake. *Variation in working memory.* Oxford University Press, 2008.

[73] C. Couto, J. E. Montandon, C. Silva, and M. T. Valente. Static correspondence and correlation between field defects and warnings reported by a bug finding tool. *Software Quality Journal*, 21:241–257. 2011.

[74] J. R. Crawford. Introduction to the assessment of attention and executive functioning. *Neuropsychological rehabilitation*, 8(3):209–211. Taylor & Francis, 1998.

[75] W. Cunningham. The wycash portfolio management system. *OOPS Messenger*, 4(2):29–30. 1993.

[76] F. Deissenbock and M. Pizka. Concise and consistent naming. In *IWPC*, 2005.

[77] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia. A developer centered bug prediction model. *IEEE Transactions on Software Engineering*, 44(1):5–24. IEEE, 2017.

[78] T. J. DiCiccio and B. Efron. Bootstrap confidence intervals. *Statistical science*, pages 189–212. JSTOR, 1996.

[79] D. Dig. A refactoring approach to parallelism. *IEEE Software*, 28(1):17–22. 2011.

[80] J. Dorn. A general software readability model. Master's thesis, University of Virginia, Charlottesville, VA, USA, 2012.

[81] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: improving software quality and reducing risk.* Pearson Education, 2007.

[82] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. Selecting empirical methods for software engineering research. In F. Shull, J. Singer, and D. I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer, 2008.

[83] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik. Confusion in code reviews: Reasons, impacts, and coping strategies. In X. Wang, D. Lo, and E. Shihab, editors, *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 49–60. IEEE, 2019.

[84] S. J. Ebisch, D. Mantini, R. Romanelli, M. Tommasi, M. G. Perrucci, G. L. Romani, R. Colom, and A. Saggino. Long-range functional interactions of anterior insula and medial frontal cortex are differently modulated by visuospatial and inductive reasoning tasks. *Neuroimage*, 78:426–438. Elsevier, 2013.

[85] B. A. Eriksen and C. W. Eriksen. Effects of noise letters upon the identification of a target letter in a nonsearch task. *Perception & psychophysics*, 16(1):143–149. Springer, 1974.

[86] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23. May 2000.

[87] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. O. Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings of the 26th Conference on Program Comprehension, ICPC*, pages 286–296, 2018.

[88] S. Fakhoury, D. Roy, S. A. Hassan, and V. Arnaoudova. Improving source code readability: Theory and practice. In *IEEE International Conference on Program Comprehension*, page To appear. IEEE, 2019.

[89] J. Fan, B. D. McCandliss, T. Sommer, A. Raz, and M. I. Posner. Testing the efficiency and independence of attentional networks. *Journal of cognitive neuroscience*, 14(3):340–347. MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info . . . , 2002.

[90] R. P. Fellows and M. Schmitter-Edgecombe. Symbol digit modalities test: Regression-based normative data and clinical utility. *Archives of Clinical Neuropsychology*, 35(1):105–115. Oxford University Press, 2020.

[91] M. Fischer, M. Pinzger, and H. C. Gall. Populating a release history database from version control and bug tracking systems. In *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*, page 23, 2003.

[92] R. A. Fisher. On the interpretation of χ 2 from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society*, 85(1):87–94. JSTOR, 1922.

[93] R. Flesch. How to write plain english: A book for la wyers and consumers. HeinOnline, 2014.

[94] B. Floyd, T. Santander, and W. Weimer. Decoding the representation of code in the brain: An fmri study of code review and expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 175–186. IEEE, 2017.

[95] M. Fokaefs, N. Tsantalis, E. Stroulia, and A. Chatzigeorgiou. Jdeodorant: identification and application of extract class refactorings. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011*, pages 1037–1039. ACM, 2011.

[96] M. Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 2018.

[97] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.

[98] M. Fowler and M. Foemmel. Continuous integration. *Thought-Works*, 122:14. 2006. http://www. thoughtworks. com/Continuous Integration. pdf.

[99] M. D. Gellman and J. R. Turner. *Cognition*, pages 441–441. Springer New York, New York, NY, 2013.

[100] R. C. Gershon, M. V. Wagster, H. C. Hendrie, N. A. Fox, K. F. Cook, and C. J. Nowinski. Nih toolbox for assessment of neurological and behavioral function. *Neurology*, 80(11 Supplement 3):S2–S6. AAN Enterprises, 2013.

[101] R. B. Grady. *Practical software metrics for project management and process improvement.* Prentice-Hall, Inc., 1992.

[102] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Transactions on software engineering*, 26(7):653–661. IEEE, 2000.

[103] C. S. Green and D. Bavelier. Action-video-game experience alters the spatial resolution of vision. *Psychological science*, 18(1):88–94. SAGE Publications Sage CA: Los Angeles, CA, 2007.

[104] M. Habib and M. Besson. What do music training and musical experience teach us about brain plasticity? *Music Perception*, 26(3):279–285. University of California Press USA, 2009.

[105] J. Halberda, M. M. Mazzocco, and L. Feigenson. Individual differences in non-verbal number acuity correlate with maths achievement. *Nature*, 455(7213):665–668. Nature Publishing Group, 2008.

[106] M. Hall, N. Walkinshaw, and P. McMinn. Supervised software modularisation. In *28th IEEE International Conference on Software Maintenance, ICSM*, pages 472–481, 2012.

[107] M. H. Halstead. *Elements of software science*, volume 7 of *Operating and programming systems series*. Elsevier, 1977.

[108] F. E. Harrell Jr, with contributions from Charles Dupont, and many others. *Hmisc: Harrell Miscellaneous*, 2017. R package version 4.0-3.

[109] Z. He, F. Shu, Y. Yang, M. Li, and Q. Wang. An investigation on the feasibility of cross-project defect prediction. *Automated Software Engineering*, 19(2):167–199. Springer, 2012.

[110] B. Henderson-Sellers, L. L. Constantine, and I. M. Graham. Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design). *Object Oriented Systems*, 3:143–158. 1996.

[111] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131. ACM New York, NY, USA, 2016.

[112] Y. Huang, X. Liu, R. Krueger, T. Santander, X. Hu, K. Leach, and W. Weimer. Distilling neural representations of data structure manipulation using fmri and fnirs. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 396–407. IEEE, 2019.

[113] M. Hubert and E. Vandervieren. An adjusted boxplot for skewed distributions. *Computational statistics & data analysis*, 52(12):5186–5201. Elsevier, 2008.

[114] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proceedings of 16th International conference on Software engineering*, pages 191–200. IEEE, 1994.

[115] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. Germán, and D. E. Damian. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering*, 21(5):2035–2071. 2016.

[116] Z. Karas, A. Jahn, W. Weimer, and Y. Huang. Connecting the dots: rethinking the relationship between code and prose writing with functional connectivity. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 767–779, 2021.

[117] N. Kasto and J. Whalley. Measuring the difficulty of code comprehension tasks using software metrics. In *Proceedings of the Fifteenth Australasian Computing Education Conference-Volume 136*, pages 59–65, 2013.

[118] D. Kawrykow and M. P. Robillard. Improving api usage through automatic detection of redundant code. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 111–122, 2009.

[119] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the 20th International Symposium on Foundations of Software Engineering*, Nov. 2012.

[120] M. Kim, T. Zimmermann, and N. Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *Software Engineering, IEEE Transactions on*, 40(7):633–649. July 2014.

[121] S. Kim and M. D. Ernst. Which warnings should I fix first? In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 45–54, 2007.

[122] S. Kim, T. Zimmermann, E. J. W. Jr., and A. Zeller. Predicting faults from cached history. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 489–498, 2007.

[123] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12):971–987. IEEE, 2006.

[124] J. D. Koen, M. Aly, W.-C. Wang, and A. P. Yonelinas. Examining the causes of memory strength variability: Recollection, attention failure, or encoding variability? *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 39(6):1726. American Psychological Association, 2013.

[125] G. P. Krishnan and N. Tsantalis. Unification and refactoring of clones. In *Proceedings of the IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering*, pages 104–113, 2014.

[126] R. Krueger, Y. Huang, X. Liu, T. Santander, W. Weimer, and K. Leach. Neurological divide: an fmri study of prose and code writing. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 678–690. IEEE, 2020.

[127] A. Kuhn, S. Ducasse, and T. Gîrba. Semantic clustering: Identifying topics in source code. *Information and Software Technology*, 49(3):230–243. 2007.

[128] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 113–122, 2011.

[129] D. Lawrie, H. Feild, and D. Binkley. Syntactic identifier conciseness and consistency. In *SCAM'06*, pages 139–148, 2006.

[130] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *ICPC'06*, 2006.

[131] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Effective identifier names for comprehension and memory. *ISSE*, 3(4):303–318. 2007.

[132] T. Lee, J. Lee, and H. P. In. Effect analysis of coding convention violations on readability of post-delivered code. *IEICE Transactions*, 98-D(7):1286–1296. 2015.

[133] A. M. Leitão. Detection of redundant code using r2d2. *Software Quality Journal*, 12(4):361–382. Dec. 2004.

[134] J.-L. Letouzey and T. Coq. The sqale analysis model: An analysis model compliant with the representation condition for assessing the quality of software source code. In *Second International Conference on Advances in System Testing and Validation Lifecycle*, pages 43–48. IEEE, 2010.

[135] Z. Li, X.-Y. Jing, and X. Zhu. Progress on approaches to software defect prediction. *Iet Software*, 12(3):161–175. IET, 2018.

[136] R. Likert. A technique for the measurement of attitudes. *Archives of psychology*. 1932.

[137] L. Lilienthal, E. Tamez, J. T. Shelton, J. Myerson, and S. Hale. Dual n-back training increases the capacity of the focus of attention. *Psychonomic bulletin & review*, 20(1):135–141. Springer, 2013.

[138] B. Lin, C. Nagy, G. Bavota, and M. Lanza. On the impact of refactoring operations on code naturalness. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 594–598, 2019.

[139] B. Lin, S. Scalabrino, A. Mocci, R. Oliveto, G. Bavota, and M. Lanza. Investigating the use of code analysis and nlp to promote a consistent usage of identifiers. In *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 81–90. IEEE, 2017.

[140] M. Mahmoudi, S. Nadi, and N. Tsantalis. Are refactorings to blame? an empirical study of refactorings in merge conflicts. In *26th IEEE International Conference*

on Software Analysis, Evolution and Reengineering, SANER 2019, pages 151–162, 2019.

[141] E. d. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik. An empirical study on the removal of self-admitted technical debt. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 238–248. IEEE, 2017.

[142] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60. JSTOR, 1947.

[143] O. Maqbool and H. A. Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11):759–780. 2007.

[144] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proc. of 21st IEEE ICSM*, pages 133–142. IEEE CS Press, 2005.

[145] A. Marcus, D. Poshyvanyk, and R. Ferenc. Using the conceptual cohesion of classes for fault prediction in object-oriented systems. *IEEE Trans. Software Eng.*, 34(2):287–300. 2008.

[146] R. C. Martin. *Clean Code: A Handbook of Agile Sofware Craftsmanship*. Prentice Hall, 2009.

[147] R. L. Matchock and J. T. Mordkoff. Chronotype and time-of-day influences on the alerting, orienting, and executive components of attention. *Experimental brain research*, 192(2):189–198. Springer, 2009.

[148] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320. Dec. 1976.

[149] J. Micco. Flaky tests at google and how we mitigate them. *Online] https://testing. googleblog. com/2016/05/flaky-tests-at-google-and-how-we. html.* 2016.

[150] S. Misra and I. Akman. Comparative study of cognitive complexity measures. In *2008 23rd International Symposium on Computer and Information Sciences*, pages 1–4. IEEE, 2008.

[151] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. L. Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36:20–36. 2010.

[152] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. Balancing agility and formalism in software engineering. chapter A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, pages 252–266. 2008.

[153] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *30th International Conference on Software Engineering (ICSE 2008)*, pages 181–190, 2008.

[154] N. Munaiah, F. Camilo, W. Wigham, A. Meneely, and M. Nagappan. Do bugs foreshadow vulnerabilities? an in-depth study of the chromium project. *Empirical Software Engineering*, 22(3):1305–1347. 2017.

[155] E. Mundy and C. K. Gilmore. Children's mapping between symbolic and non-symbolic representations of number. *Journal of experimental child psychology*, 103(4):490–502. Elsevier, 2009.

[156] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the elipse ide? *IEEE software*, 23(4):76–83. IEEE, 2006.

[157] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE software*, 25(5):38–44. IEEE, 2008.

[158] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. *Transactions on Software Engineering*, 38(1):5–18. 2011.

[159] M. Musso, E. Kyndt, E. Cascallar, and F. Dochy. Predicting mathematical performance: The effect of cognitive processes and self-regulation factors. *Education Research International*, 2012. Hindawi, 2012.

[160] S. Nour, E. Struys, and H. Stengers. Attention network in interpreters: The role of training and experience. *Behavioral Sciences*, 9(4):43. Multidisciplinary Digital Publishing Institute, 2019.

[161] Y. Oded. Biofeedback-based mental training in the military—the "mental gym™" project. *Biofeedback*, 39(3):112–118. Association for Applied Physiology and Biofeedback 10200 West 44th Ave., No . . . , 2011.

[162] D. Oliveira, M. Rosenthal, N. Morin, K.-C. Yeh, J. Cappos, and Y. Zhuang. It's the psychology stupid: how heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 296–305, 2014.

[163] D. S. Oliveira, T. Lin, M. S. Rahman, R. Akefirad, D. Ellis, E. Perez, R. Bobhate, L. A. DeLong, J. Cappos, and Y. Brun. API blindspots: Why experienced developers write vulnerable code. In *Fourteenth Symposium on Usable Privacy and Security ({SOUPS} 2018)*, pages 315–328, 2018.

[164] A. Oram and G. Wilson. *Beautiful Code: Leading Programmers Explain How They Think*. O'reilly, 2007.

[165] M. Paixão, M. Harman, Y. Zhang, and Y. Yu. An empirical study of cohesion and coupling: Balancing optimization and disruption. *IEEE Trans. Evolutionary Computation*, 22(3):394–414. 2018.

[166] F. Palomba, G. Bavota, M. D. Penta, F. Fasano, R. Oliveto, and A. D. Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188–1221. 2018.

[167] F. Palomba, A. D. Lucia, G. Bavota, and R. Oliveto. Anti-pattern detection: Methods, challenges, and open issues. *Advances in Computers*, 95:201–238. 2015.

[168] F. Palomba, D. A. Tamburri, F. A. Fontana, R. Oliveto, A. Zaidman, and A. Serebrenik. Beyond technical aspects: How do community smells influence the intensity of code smells? *IEEE Transactions on Software Engineering.* IEEE, 2018.

[169] J. Pantiuchina, G. Bavota, M. Tufano, and D. Poshyvanyk. Towards just-in-time refactoring recommenders. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018*, pages 312–315, 2018.

[170] J. Pantiuchina, M. Lanza, and G. Bavota. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 80–91. IEEE, 2018.

[171] J. Pantiuchina, F. Zampetti, S. Scalabrino, V. Piantadosi, R. Oliveto, G. Bavota, and M. D. Penta. Why developers refactor source code: A mining-based study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(4):1–30. ACM New York, NY, USA, 2020.

[172] M. C. Passolunghi, B. Vercelloni, and H. Schadee. The precursors of mathematics learning: Working memory, phonological ability and numerical competence. *Cognitive development*, 22(2):165–184. Elsevier, 2007.

[173] N. Peitek, J. Siegmund, S. Apel, C. Kästner, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann. A look into programmers' heads. *IEEE Transactions on Software Engineering*, 46(4):442–462. IEEE, 2018.

[174] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman. An empirical investigation of how and why developers rename identifiers. In *Proceedings of the 2Nd International Workshop on Refactoring*, pages 26–33, 2018.

[175] V. Piantadosi, F. Fierro, S. Scalabrino, A. Serebrenik, and R. Oliveto. How does code readability change during software evolution? *Empirical Software Engineering*, 25(6):5374–5412. Springer, 2020.

[176] V. Piantadosi, F. Fierro, S. Scalabrino, A. Serebrenik, and R. Oliveto. Replication package. `https://github.com/stakelab/replication-readability-evolution`, 2020.

[177] V. Piantadosi, S. Scalabrino, A. Serebrenik, N. Novielli, and R. Oliveto. Do attention and memory explain the performance of software developers? *Submitted at Empirical Software Engineering.* Springer.

[178] M. I. Posner. Orienting of attention. *Quarterly journal of experimental psychology*, 32(1):3–25. SAGE Publications Sage UK: London, England, 1980.

[179] M. I. Posner and S. E. Petersen. The attention system of the human brain. *Annual review of neuroscience*, 13(1):25–42. Annual Reviews 4139 El Camino Way, PO Box 10139, Palo Alto, CA 94303-0139, USA, 1990.

[180] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov. Dual ecological measures of focus in software development. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 452–461. IEEE, 2013.

[181] D. Posnett, A. Hindle, and P. Devanbu. A simpler model of software readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 73–82. ACM, 2011.

[182] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, pages 91–100. IEEE Computer Society, 2014.

[183] K. Praditwong, M. Harman, and X. Yao. Software module clustering as a multi-objective search problem. *IEEE Transactions on Software Engineering*, 37(2):264–282. 2011.

[184] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526. IEEE, 2005.

[185] F. Rahman, D. Posnett, and P. Devanbu. Recalling the" imprecision" of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[186] V. Rajlich and P. Gosavi. Incremental change in object-oriented programming. *IEEE Softw.*, 21(4):62–69. IEEE Computer Society Press, July 2004.

[187] M. D. Rapport, M. J. Kofler, R. M. Alderson, T. M. Timko Jr, and G. J. DuPaul. Variability of attention processes in adhd: Observations from the classroom. *Journal of Attention Disorders*, 12(6):563–573. Sage Publications Sage CA: Los Angeles, CA, 2009.

[188] R. H. Rasch and H. L. Tosi. Factors affecting software developers' performance: An integrated approach. *MIS quarterly*, pages 395–413. JSTOR, 1992.

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

[190] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato. The role of experience and ability in comprehension tasks supported by UML stereotypes. In *Proc. of 29th ICSE*, pages 375–384. IEEE Computer Society, 2007.

[191] F. Ricca, M. Di Penta, M. Torchiano, P. Tonella, and M. Ceccato. How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments. *IEEE Transactions on Software Engineering*, 36(1):96–118. IEEE, 2009.

[192] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *2012 34th International Conference on Software Engineering (ICSE)*, pages 255–265. IEEE, 2012.

[193] B. Rosner. *Fundamentals of Biostatistics.* Brooks/Cole, Boston, MA, 7th edition edition, 2011.

[194] C. K. Roy. Large scale clone detection, analysis, and benchmarking: An evolutionary perspective. In *12th IEEE International Workshop on Software Clones, IWSC*, 2018.

[195] E. Roy. *Cognitive Function*, pages 448–449. Springer New York, New York, NY, 2013.

[196] P. Runeson, M. Alexandersson, and O. Nyholm. Detection of duplicate defect reports using natural language processing. In *Proc. of 29th ICSE*, pages 499–510, 2007.

[197] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. Automatically assessing code understandability: How far are we? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 417–427. IEEE Press, 2017.

[198] S. Scalabrino, G. Bavota, C. Vendome, D. Poshyvanyk, and R. Oliveto. Automatically assessing code understandability. *IEEE Transactions on Software Engineering.* IEEE, 2019.

[199] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6). 2018.

[200] S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto. Improving code readability models with textual features. In *Program Comprehension (ICPC), 2016 IEEE 24th International Conference on*, pages 1–10. IEEE, 2016.

[201] D. L. Schacter, A. D. Wagner, and R. L. Buckner. Memory systems of 1999. Oxford University Press, 2000.

[202] E. G. Schellenberg. Music lessons enhance iq. *Psychological science*, 15(8):511–514. SAGE Publications Sage CA: Los Angeles, CA, 2004.

[203] Z. Sharafi, Y. Huang, K. Leach, and W. Weimer. Toward an objective measure of developers' cognitive activities. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30(3):1–40. ACM New York, NY, USA, 2021.

[204] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures.* Chapman & Hall/CRC, 4 edition, 2007.

[205] B. Shneiderman and R. Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3):219–238. Springer, 1979.

[206] J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. Measuring and modeling programming experience. *Empirical Software Engineering*, 19(5):1299–1334. 2014.

[207] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 140–150, 2017.

[208] D. Silva, N. Tsantalis, and M. T. Valente. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 858–870. ACM, 2016.

[209] L. Sloan. *Learn about Spearman's Rank-order Correlation Coefficient in SPSS with Data from the General Social Survey (2012).* SAGE Publications, 2015.

[210] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136. ACM, 2006.

[211] D. Spadini, F. Palomba, T. Baum, S. Hanenberg, M. Bruntink, and A. Bacchelli. Test-driven code review: an empirical study. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, pages 1061–1072, 2019.

[212] C. Spearman. The proof and measurement of association between two things. Appleton-Century-Crofts, 1961.

[213] D. Spencer. *Card sorting: Designing usable categories.* Rosenfeld Media, 2009.

[214] D. Spinellis, P. Louridas, and M. Kechagia. The evolution of C programming practices: a study of the unix operating system 1973-2015. In L. K. Dillon, W. Visser, and L. Williams, editors, *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 748–759. ACM, 2016.

[215] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *Proceedings of the 5th International Workshop on Software Quality*, pages 10–. IEEE Computer Society, 2007.

[216] G. Szoke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimóthy. Bulk fixing coding issues and its effects on software quality: Is it worth refactoring? In *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 95–104. IEEE, 2014.

[217] A. Takang, P. Grubb, and R. Macredie. The effects of comments and identifier names on program comprehensibility: an experiential study. *Journal of Programming Languages*, 4(3):143–167. 1996.

[218] M. Thongmak and P. Muenchaisri. Measuring understandability of aspect-oriented code. In *International Conference on Digital Information and Communication Technology and Its Applications*, pages 43–54. Springer, 2011.

[219] M. K. Thota, F. H. Shajin, P. Rajesh, et al. Survey on software defect prediction techniques. *International Journal of Applied Science and Engineering*, 17(4):331–344. Chaoyang University of Technology, 2020.

[220] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner, and B. Vasilescu. Automatically assessing code understandability reanalyzed: combined metrics matter. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 314–318. ACM, 2018.

[221] N. Tsantalis and A. Chatzigeorgiou. Identification of move method refactoring opportunities. *IEEE Transactions on Software Engineering*, 35(3):347–367. 2009.

[222] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, pages 483–494, 2018.

[223] N. Tsantalis, D. Mazinanian, and S. Rostami. Clone refactoring with lambda expressions. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 60–70, 2017.

[224] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088. IEEE, 2017.

[225] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019*, pages 25–36, 2019.

[226] P. A. Tun and M. E. Lachman. Telephone assessment of cognitive function in adulthood: the brief test of adult cognition by telephone. *Age and Ageing*, 35(6):629–632. Oxford University Press, 2006.

[227] A. Van Deursen, L. Moonen, A. Bergh, and G. Kok. Refactoring test code. Technical report, Amsterdam, 2001.

[228] B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand. By no means: a study on aggregating software metrics. In G. Concas, E. D. Tempero, H. Zhang, and M. D. Penta, editors, *Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics, WETSoM 2011, Waikiki, Honolulu, HI, USA, May 24, 2011*, pages 23–26. ACM, 2011.

[229] C. Vassallo, G. Grano, F. Palomba, H. Gall, and A. Bacchelli. A large-scale empirical exploration on refactoring activities in open source software projects. *Science of Computer Programming*, 180(1):1–15. 2019.

[230] H. Wang, J. Fan, and Y. Yang. Toward a multilevel analysis of human attentional networks. In *Proceedings of the Annual Meeting of the Cognitive Science Society*, 2004.

[231] Y. Wang. What motivate software engineers to refactor source code? evidences from professional developers. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 413 –416, 2009.

[232] B. Weaver, M. Bédard, and J. McAuliffe. Evaluation of a 10-minute version of the attention network test. *The Clinical Neuropsychologist*, 27(8):1281–1299. Taylor & Francis, 2013.

[233] B. Weaver, M. Bedard, J. McAuliffe, and M. Parkkari. Using the attention network test to predict driving test scores. *Accident Analysis & Prevention*, 41(1):76–83. Elsevier, 2009.

[234] W. Wei, H. Yuan, C. Chen, and X. Zhou. Cognitive correlates of performance in advanced mathematics. *British Journal of Educational Psychology*, 82(1):157–181. Wiley Online Library, 2012.

[235] E. J. Weyuker. Evaluating software complexity measures. *IEEE transactions on Software Engineering*, 14(9):1357–1365. IEEE, 1988.

[236] F. Wilcoxon. Individual comparisons by ranking methods. biometrics bulletin 1, 6 (1945), 80–83. *URL http://www. jstor. org/stable/3001968*. 1945.

[237] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308. IEEE, 2013.

[238] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. *Software: Practice and Experience*, 28(4):347–369. Wiley Online Library, 1998.

[239] E. Woumans, E. Ceuleers, L. Van der Linden, A. Szmalec, and W. Duyck. Verbal and nonverbal cognitive control in bilinguals and interpreters. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 41(5):1579. American Psychological Association, 2015.

[240] F. Zampetti, C. Noiseux, G. Antoniol, F. Khomh, and M. D. Penta. Recommending when design technical debt should be self-admitted. In *2017 IEEE International Conference on Software Maintenance and Evolution, ICSME 2017, Shanghai, China, September 17-22, 2017*, pages 216–226. IEEE Computer Society, 2017.

[241] F. Zampetti, A. Serebrenik, and M. D. Penta. Was self-admitted technical debt removal a real removal?: an in-depth perspective. In A. Zaidman, Y. Kamei, and E. Hill, editors, *Proceedings of the 15th International Conference on Mining*

Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018, pages 526–536. ACM, 2018.

[242] F. Zhang, F. Khomh, Y. Zou, and A. E. Hassan. An empirical study on factors impacting bug fixing time. In *2012 19th Working conference on reverse engineering*, pages 225–234. IEEE, 2012.

[243] Y. Zhang, G. Huang, X. Liu, W. Zhang, H. Mei, and S. Yang. Refactoring android java code for on-demand computation offloading. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, pages 233–248, 2012.

[244] Y. Zhao, A. Serebrenik, Y. Zhou, V. Filkov, and B. Vasilescu. The impact of continuous integration on other software development practices: a large-scale empirical study. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 60–71. IEEE Press, 2017.

[245] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 91–100, 2009.