Noname manuscript No.
(will be inserted by the editor)

How Does Code Readability Change During Software Evolution?

Valentina Piantadosi · Fabiana Fierro · Simone Scalabrino · Alexander Serebrenik · Rocco Oliveto

Received: date / Accepted: date

Abstract Code reading is one of the most frequent activities in software maintenance. Such an activity aims at acquiring information from the code and, thus, it is a prerequisite for program comprehension: developers need to read the source code they are going to modify before implementing changes. As the code changes, so does its readability; however, it is not clear yet *how* code readability changes during software evolution.

To understand how code readability changes when software evolves, we studied the history of 25 open source systems. We modeled code readability evolution by defining four states in which a file can be at a certain point of time (*non-existing*, *other-name*, *readable*, and *unreadable*). We used the data gathered to infer the probability of transitioning from one state to another one. In addition, we also manually checked a significant sample of transitions to compute the performance of the state-of-the-art readability prediction model we used to calculate the transition probabilities. With this manual analysis, we found that the tool correctly classifies all the transitions in the majority of the cases, even if there is a loss of accuracy compared to the single-version readability estimation. Our results show that most of the source code files are created readable. Moreover, we observed that only a minority of the commits change the readability state.

Finally, we manually carried out qualitative analysis to understand what makes code unreadable and what developers do to prevent this. Using our results we propose some guidelines (i) to reduce the risk of code readability erosion and (ii) to promote best practices that make code readable.

Keywords Code Readability · Software Evolution · Mining Software Repositories

V. Piantadosi and F. Fierro and S. Scalabrino and R. Oliveto
University of Molise, Italy
E-mail:
{valentina.piantadosi, simone.scalabrino, rocco.oliveto}@unimol.it
f.fierro1@studenti.unimol.it

Alexander Serebrenik
Eindhoven University of Technology, The Netherlands
E-mail: a.serebrenik@tue.nl

1 Introduction

"Readability" is a fundamental and highly desirable property of the source code. Code reading is the very first step during incremental change (Bennett and Rajlich, 2000; Rajlich and Gosavi, 2004), which is required to perform concept location, impact analysis and the corresponding change implementation/propagation. Assuring source code readability becomes an imperative in modern open source software development due to its collaborative and geographically distributed character. Erlikh (2000) 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.

Several facets have been reported as components that contribute to making code readable (Martin, 2009; Oram and Wilson, 2007; Beck, 2007). Such components include complexity, usage of design concepts, formatting, source code lexicon, and visual aspects (*i.e.*, syntax highlighting). Indeed, previous work provide empirical evidence that structural (Buse and Weimer, 2010; Posnett et al., 2011), visual (Dorn, 2012), and textual (Scalabrino et al., 2016, 2018) 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. (2015) explored 210 open source Java projects in order 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. (2016) 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 for other programming languages.

In this paper, we take a closer look at how code readability changes in the evolution of software systems. First, we conducted a survey with 122 developers to understand to what extent code readability is important to them, and we found that the vast majority (∼83.8%) often take code readability into account when writing code. 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 (Scalabrino et al., 2018) 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 snapshot: 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, involved in ∼83k commits. We studied the evolution of readability at 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 project 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 also 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 low the number of unreadable files.

The remainder of the paper is organized as follows. Section 2 discusses the related literature. In Section 3 we report a motivating study for this work through which we aim at understanding to what extent developers care about code readability. Section 4 presents the model we used to describe the readability evolution of a file. In Section 5 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 6 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 discusses the threats to validity of the studies and Section 8 concludes the paper.

2 Related Works

Quality of the source code has been extensively studied in the literature starting from the 1960 when first software metrics have been proposed, like *Lines Of Code*, *McCabe's Cyclomatic Complexity* (McCabe, 1976) and *Halstead's metrics* (Halstead, 1977).

While none of them explicitly refers to readability, many of them include related notions of maintanability or understandability (McCall et al., 1977; Boehm et al., 1978; Grady, 1992). This being said, in ISO 9126 *understandability* takes the perspective of the end-user rather than the one of the developer as, for instance, in the case of the earlier model by Boehm et al. (1978) and SQALE (Letouzey and Coq, 2010). Given the extensive research on software quality, providing a complete overview is not possible and we focus on readability from here on.

2.1 Code Readability

Previous works focused on automatic assessment of code readability (Buse and Weimer, 2008, 2010; Posnett et al., 2011; Dorn, 2012; Scalabrino et al., 2018). 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. In order 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 (Buse and Weimer, 2008; Dorn, 2012; Scalabrino et al., 2016). All such datasets 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.

Buse and Weimer (2008, 2010) designed the first readability model based on structural features, such as line and identifier length, number of loops, identifiers, keywords, parenthesis, arithmetic and comparison operators, and so on. Posnett et al. (2011) defined a simpler model, similar to the ones used to estimate the readability of text in natural language. Such a model uses just three features: lines of code (LOC), entropy and volume. Dorn (2012) 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.

Scalabrino et al. (2016, 2018) 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.

Pantiuchina et al. (2018) 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. Fakhoury et al. (2019) recently reported a similar result. This effect could be due to the fact that even if a change improves readability, the improvement might be small with respect to the size of the changed class, and so this makes explainable the difference in the metric.

The studies most related to the one we report in this paper are the ones by Lee et al. (2015) and Spinellis et al. (2016). Lee et al. (2015) 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 show 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, Spinellis et al. (2016) observed that the readability of Unix gradually improved. However, they also highlight that there is insufficient evidence to claim that readability is still increasing.

We build on such studies and we extend them (i) by re-assessing the accuracy of readability models in assessing readability transitions, (ii) by defining a readability evolution model to describe such a process, and (iii) by conducting a large study in which we consider 25 software systems.

2.2 Software/Code Understandability

Code readability represents how easily information is conveyed to the developer. Several works focused, instead, on measuring an apparently 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. (2004) defined a measure of understandability in OSS projects. Specifically, this measure is computed on the history of 19 open source projects in the following way: (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 growns.

Misra and Akman (2008) used the properties proposed by Weyuker (1988) to compare existing cognitive measures and they proposed some measures, *i.e.*, the Cognitive Weight Complexity Measure (CWCM). The authors assigned weights to software components through the analysis of their control structures.

Thongmak and Muenchaisri (2011) evaluated the understandability of aspect-oriented software through aspect-oriented software dependence graphs.

Subsequently, Chen et al. (2016) explored the COCOMO II Software Understandability factors 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.

Scalabrino et al. (2017, 2019) performed an in-depth analysis of 121 metrics divided in three categories: (i) new code-related, (ii) documentation-related, and (iii) developer-related. The authors 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. The interviews with 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. (2018) performed a reanalysis of the dataset by Scalabrino et al. (2017) by combining metrics with statistical modeling techniques. They showed that the combination of metrics improve the assessment of code understandability, as also reported by Scalabrino et al. (2019).

3 Motivating Study: Survey of Developer Behavior

Any developer would clearly prefer working on readable code rather than on unreadable code. 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 (Pantiuchina et al., 2018; Fakhoury et al., 2019): 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. Therefore, before trying to define a

model for describing code readability evolution, we wanted to understand if this is a problem worth of investigation. For this reason, we run the motivating study presented in this section.

3.1 Survey Design

The *goal* of our motivating 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 motivating study we want to answer to 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 main study (Section 6).

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, used the Likert scale (Likert, 1932) 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 Siegmund et al. (2014)), (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.

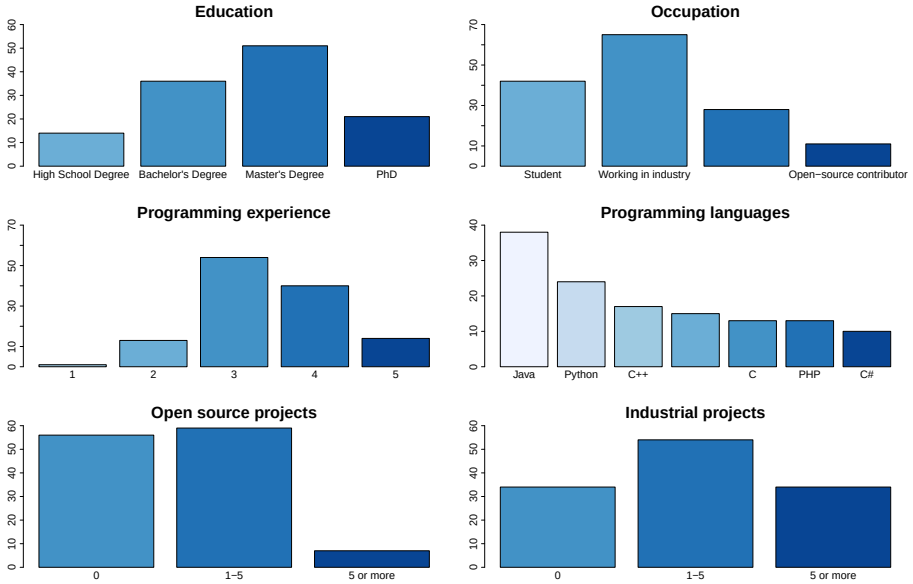

Fig. 1: Distribution of the answer to demographic questions.

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

3.2 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 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 an high school degree (14, *i.e.*, 11.5%) instead. The five most popular programming 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 Figure 2 and Figure 3 the distribution of the answers to the four questions. We use two different colors to highlight the differences between

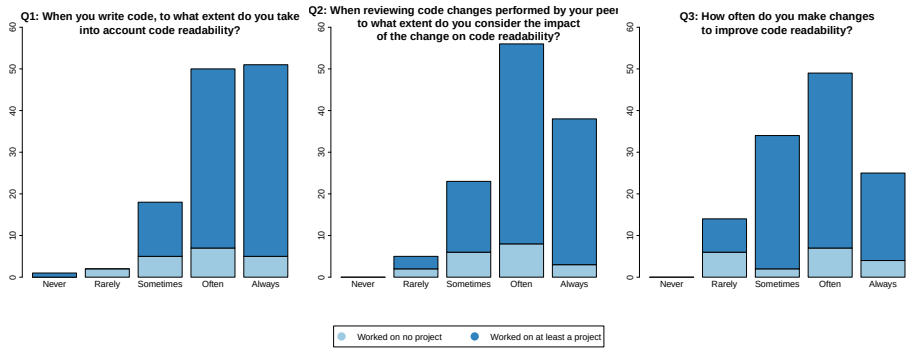

Fig. 2: Distribution of the answers to Q_1, Q_2, and Q_3.

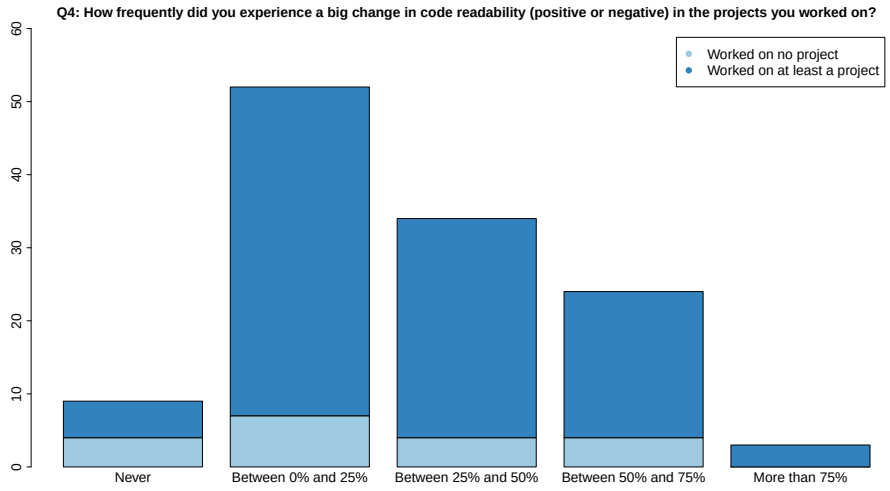

Fig. 3: Distribution of the answers to Q_4.

developers who did not contribute to any collaborative project and developers who contributed to at least an open source/industrial project. Figure 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.

Interestingly, 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. This suggests that developers understand the importance of readability, but they consider it not

a priority while peer-reviewing code. This can have many causes, *e.g.*, it could happen for social reasons (Palomba et al., 2018).

Finally, Figure 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 Figure 2 and Figure 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 6.

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

4 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 (Buse and Weimer, 2010; Scalabrino et al., 2018), 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 Section 2, 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 Section 5 and Section 6 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 bag 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

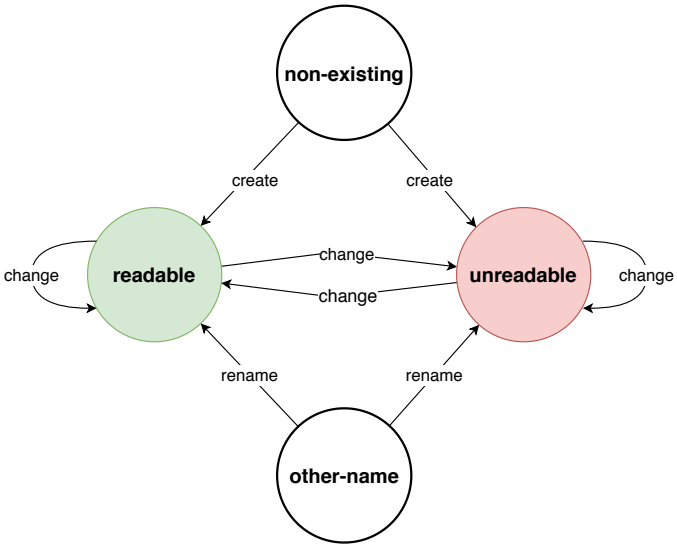

Fig. 4: States of a source code file.

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 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 4 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

Table 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**

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 6.

5 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 4 is a class. This preliminary study will allow us to better frame the main study, reported in Section 6.

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 from the dataset we introduce in this paper: 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.

5.1 Data Collection

The context of our study is constituted by archival data (Runeson et al., 2012). Specifically, we have studied the history of 25 Java open source projects. We report in Table 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 (Easterbrook et al., 2008). 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* $\rightarrow$ {*readable*/*unreadable*}; when a file was modified, we measured its readability: in such cases, we added a new transition {*readable*/*unreadable*} $\rightarrow$ {*readable*/*unreadable*}; when a file was renamed, we introduced a new transition *other-name* $\rightarrow$ {*readable*/*unreadable*}.

We chose the tool by Scalabrino et al. (2018) 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. (2018)).

All the approaches available in the literature, including the one we used, were validated on small snippets (*e.g.*, methods) (Buse and Weimer, 2008, 2010; Posnett et al., 2011; Dorn, 2012; Scalabrino et al., 2016, 2018). 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 (Vasilescu et al., 2011a). 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 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. (2018) with all the Java snippets and these are from the union of three readability datasets currently available (Buse and Weimer, 2010; Dorn, 2012; Scalabrino et al., 2016), 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 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.

5.2 Experimental Procedure

To answer $\mathbf{RQ}_1$, we used the dataset provided by Pantiuchina et al. (2018). 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}* $\rightarrow$ *{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* $\rightarrow$ *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* $\rightarrow$ *readable*, this agrees with what developers claimed; on the other hand, if the predicted transition is *readable* $\rightarrow$ *readable* it can still be true that there was an improvement in readability; even if the predicted transition is *unreadable* $\rightarrow$ *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 (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

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

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. (2018), 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 $\mathbf{RQ}_2$, 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, (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 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 5.3 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:

$$precision_t = \frac{TP_t}{TP_t + FP_t}$$

$$recall_t = \frac{TP_t}{TP_t + FN_t}$$

$$F_t = 2 * \frac{precision_t * recall_t}{precision_t + recall_t},$$

Table 2: Datasets used in the first study.

Research Question	Dataset	No. of transitions
RQ_1	Pantiuchina et al. (2018)	1,282
RQ_2	Significant subset of our dataset	271

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 2 a summary of the dataset we used to answer each research question in this first study. We provide a replication package (Piantadosi et al., 2020) with (i) the dataset we built and (ii) the data used to answer RQ_1 and RQ_2.

5.3 Empirical Study Results

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

5.3.1 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. (2018) 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

[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 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 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%

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. (2018), 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 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. (2018) is not reliable when the output is in the range [0.416, 0.600].

5.3.2 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, which result is in line with the accuracy reported in the original study (*i.e.*, ∼84%).

As for transitions, we report in Table 3 the *confusion matrix* for the 8-class categorization problem. Also, in Table 4 we report precision, recall, and F-measure obtained on the samples we manually evaluated. We found that the tool by Scalabrino et al. (2018) 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 is due to the fact that this is a classification problem involving eight classes, which is more difficult than a classification problem involving just two classes (*i.e.*, *readable* and *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.

Summary of RQ$_2$. There is a loss of accuracy when using readability prediction tools for transitions instead of single versions.

6 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 3, with this research question we also want to verify if there is a match between what people *say* with what people *do* (Easterbrook et al., 2008; Ebert et al., 2019).

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.

Table 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

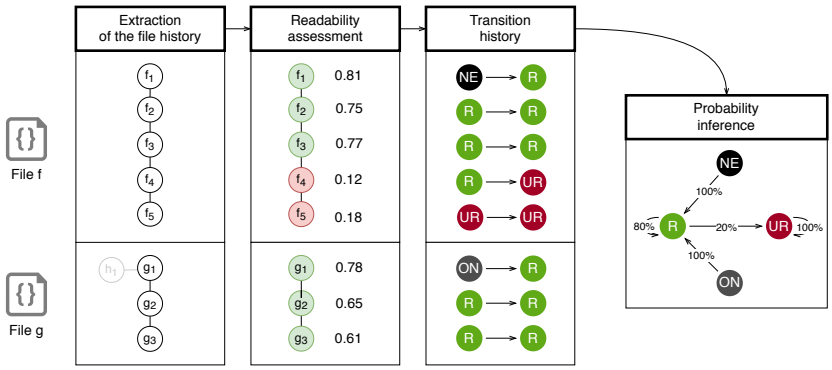

Fig. 5: The process we used to compute the transition probability of a project.

6.1 Experimental Procedure

To answer both our research questions, we used the dataset we collected (described in Section 5.1). We report in Table 5 a summary of the datasets we have used to answer each research question.

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 4. 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 5 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:

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

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

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 (DiCiccio and Efron, 1996) to compute the confidence intervals of each probability.

Specifically, we used the m-out-of-n bootstrap (Chernick, 2011), 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 $m = 0.8n$ (Chernick, 2011). 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 (1995). The p-values for correlations represent evidence against the null-hypothesis that the correlation between the ranks of the variables we study equals 0 (Sheskin, 2007). 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* $\rightarrow$ *unreadable* or *unreadable* $\rightarrow$ *readable*). Then, we divided the data we gathered in three groups:

- R^+: transitions representing an increase of code readability (*i.e.*, transitions *readable* $\rightarrow$ *unreadable*);
- R^-: transitions representing a decrease of code readability (*i.e.*, transitions *unreadable* $\rightarrow$ *readable*);
- R^0: transitions that did not result in a readability change (*i.e.*, transitions *readable* $\rightarrow$ *readable* and *unreadable* $\rightarrow$ *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 (Wilcoxon, 1945) 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 (1995) method. We also compute the effect size to quantify the magnitude of the significant differences we find. We use Cliff's delta (Cliff, 1993) 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 $\mathbf{RQ}_4$, 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);
- why readability changed (*e.g.*, comments added or long lines removed), focusing on three aspects: *visual*, *structural*, and *textual* (Scalabrino et al., 2018).

In case of disagreement, two authors performed an open discussion to resolve conflicting cases. We used the results of the analysis to formulate suggestions, available in Section 6.2, on how to avoid readability deterioration and annotations in order to improve source code readability.

In Figure 6 we summarize the process used for our two studies. Our replication package (Piantadosi et al., 2020) contains the datasets we used to answer RQ_3 and RQ_4 to foster the replicability of this study.

6.2 Empirical Study Results

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

6.2.1 RQ_3: How Often Does Readability Change?

Table 6 and Table 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

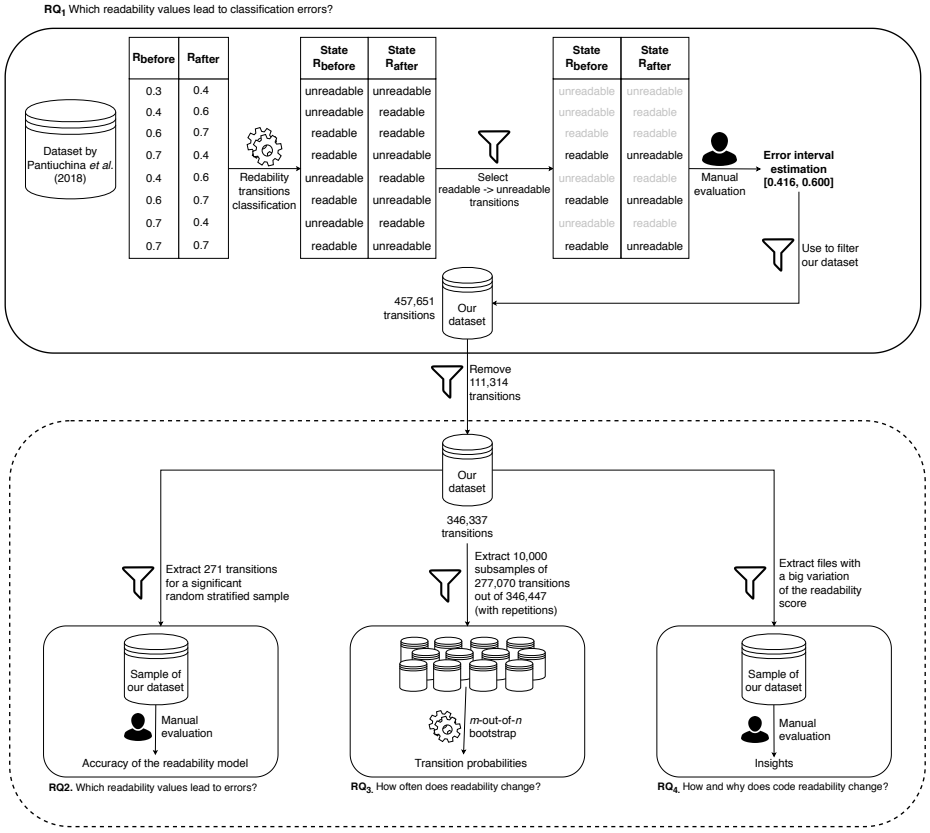

Fig. 6: Process of Study I and Study II.

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 (Bavota et al., 2015; Chatzigeorgiou and Manakos, 2014; Tufano et al., 2017; Silva et al., 2016; Maldonado et al., 2017; Zampetti et al., 2018). We provide more details about this aspect in our qualitative analysis.

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.

Table 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%]	//	//

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 8 shows the aforementioned results: only a minority (usually less than 5%) of the files changes 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 is the history of the project, 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

Table 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%]

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 9 shows the results of our analysis regarding the characteristics of readability evolution transitions for all the data-points, while we report in Table 10 and Table 11 a the results at project-level. In Figure 7 we report the boxplots of files changed, lines changed, lines added and lines removed for each group, adjusted for skewed distributions (Hubert and Vandervieren, 2008).

Table 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 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)**

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 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 $<$

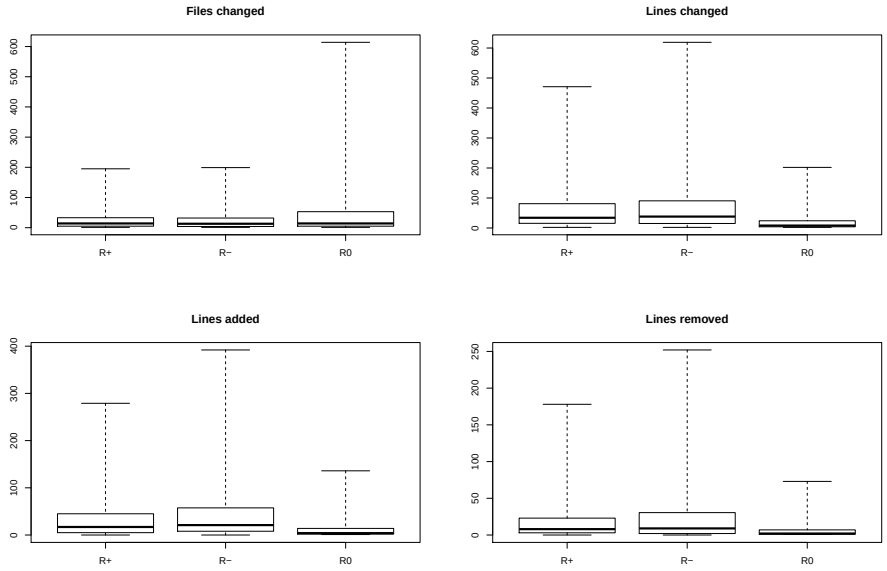

Fig. 7: Adjusted boxplots of files changed, lines changed, lines added and lines removed for each group (without outliers).

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 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 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* $\rightarrow$ *readable* or *unreadable* $\rightarrow$ *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

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

Table 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 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)	//	//	//	//

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.

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 transition 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.

6.2.2 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

[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`

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 (Pantiuchina et al., 2018), 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,

[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`

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) (Bavota et al., 2015; Chatzigeorgiou and Manakos, 2014; Tufano et al., 2017; Silva et al., 2016; Maldonado et al., 2017; Zampetti et al., 2018): changes in readability are mostly done unintentionally.

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* 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`,

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

[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`

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

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

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.

6.2.3 Discussion

The main finding of our study is that code readability rarely changes. A first hint towards this finding could be found in survey presented in Section 3, 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%).

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 (Tufano et al., 2017) and dependencies on unstable APIs (Businge et al., 2015): 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

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

(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 on (Cunningham, 1993): at the very least such technical debt should be either explicitly admitted by developers (Potdar and Shihab, 2014), or flagged by specialized tools (Zampetti et al., 2017).

- **Review 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. (2000), 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 and Perry (2005) 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 and Foemmel (2006) and Duvall et al. (2007) support commit small in the CI guidelines. Zhao et al. (2017) 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. (2019) 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 (Murphy-Hill and Black, 2008). Developers already prefer the floss-refactoring (*i.e.*, frequent refactoring steps interleaved with their regular activities). Consequently, on 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 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 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.

7.1 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 (Bird et al., 2009; Kalliamvakou et al., 2016). We made sure that we excluded personal/toy projects and repositories not used for software development (see the work of Munaiah et al. (2017) 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 (Scalabrino et al., 2018). 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 (Vasilescu et al., 2011b): 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 (Vasilescu et al., 2011a). 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 Scalabrino et al. (2018)), 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.

7.2 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 $\sim$0.416 and $\sim$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.

7.3 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.

8 Conclusion

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 paper, we introduced a descriptive model for the evolution of code readability at file level. We conducted a large empirical study to understand how and why code readability changes during software evolution. We considered the history of 25 projects, for a total of ∼83K commits. Our results show that code readability rarely changes: therefore, when a file is created *unreadable* it most likely remains *unreadable*. We also found that readability changes mostly happen because of adaptive changes that modify much code in the class. However, unreadable code is a minority of the code in most of the projects we studied. Finally, we observed that current readability models are not very well suited for classifying readability changes.

Our results suggest that well known best practices, such as making small commits and performing refactoring operations when needed, help reducing the amount of unreadable code. In addition, carefully reviewing code when new features are introduced can help reducing the quantity of unreadable code.

Our results call for the following future research directions. First, new readability prediction models specifically designed to classify changes instead of single snapshots would be necessary. Also, using a discrete-time Markov chain (non-time-homogeneous) could help defining a more fine-grained and accurate readability evolution model.

References

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

Beck K (2007) Implementation Patterns. Addison Wesley

Benjamini Y, Hochberg Y (1995) 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

Bennett KH, Rajlich VT (2000) Software maintenance and evolution: A roadmap. In: Proceedings of the Conference on The Future of Software Engineering, pp 73–87, DOI 10.1145/336512.336534

Bird C, Rigby PC, Barr ET, Hamilton DJ, Germán DM, Devanbu PT (2009) The promises and perils of mining git. In: Godfrey MW, Whitehead J (eds) Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings, IEEE Computer Society, pp 1–10

Boehm BW, Brown JR, Kaspar H, Lipow M, Macleod GJ, Merrit MJ (1978) Characteristics of Software Quality, TRW series of software technology, vol 1. Elsevier

Buse RP, Weimer WR (2008) A metric for software readability. In: Proceedings of the 2008 international symposium on Software testing and analysis, ACM, pp 121–130

Buse RP, Weimer WR (2010) Learning a metric for code readability. Software Engineering, IEEE Transactions on 36(4):546–558

Businge J, Serebrenik A, van den Brand MGJ (2015) Eclipse API usage: the good and the bad. Softw Qual J 23(1):107–141, DOI 10.1007/s11219-013-9221-3, URL https://doi.org/10.1007/s11219-013-9221-3

Capiluppi A, Morisio M, Lago P (2004) Evolution of understandability in oss projects. In: Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings., IEEE, pp 58–66

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

Chen C, Alfayez R, Srisopha K, Shi L, Boehm B (2016) Evaluating human-assessed software maintainability metrics. In: National Software Application Conference, Springer, pp 120–132

Chernick MR (2011) Bootstrap methods: A guide for practitioners and researchers, vol 619. John Wiley & Sons

Cliff N (1993) Dominance statistics: Ordinal analyses to answer ordinal questions. Psychological bulletin 114(3):494

Cunningham W (1993) The wycash portfolio management system. OOPS Messenger 4(2):29–30

DiCiccio TJ, Efron B (1996) Bootstrap confidence intervals. Statistical science pp 189–212

Dorn J (2012) A general software readability model. Master's thesis, University of Virginia, Charlottesville, VA, USA, URL http://www.cs.virginia.edu/~weimer/students/dorn-mcs-paper.pdf

Duvall PM, Matyas S, Glover A (2007) Continuous integration: improving software quality and reducing risk. Pearson Education

Easterbrook S, Singer J, Storey MA, Damian D (2008) Selecting empirical methods for software engineering research. In: Shull F, Singer J, Sjøberg DIK (eds) Guide to Advanced Empirical Software Engineering, Springer, pp 285–311

Ebert F, Castor F, Novielli N, Serebrenik A (2019) Confusion in code reviews: Reasons, impacts, and coping strategies. In: Wang X, Lo D, Shihab E (eds) 26th

IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019, IEEE, pp 49–60
Erlikh L (2000) Leveraging legacy system dollars for e-business. IT Professional 2(3):17–23, DOI 10.1109/6294.846201
Fakhoury S, Roy D, Hassan SA, Arnaoudova V (2019) Improving source code readability: Theory and practice. In: IEEE International Conference on Program Comprehension, IEEE, p To appear
Fowler M, Foemmel M (2006) Continuous integration. Thought-Works 122:14, http://www. thoughtworks. com/Continuous Integration. pdf
Grady RB (1992) Practical software metrics for project management and process improvement. Prentice-Hall, Inc.
Graves TL, Karr AF, Marron JS, Siy H (2000) Predicting fault incidence using software change history. IEEE Transactions on software engineering 26(7):653–661
Halstead MH (1977) Elements of software science, Operating and programming systems series, vol 7. Elsevier
Hubert M, Vandervieren E (2008) An adjusted boxplot for skewed distributions. Computational statistics & data analysis 52(12):5186–5201
Kalliamvakou E, Gousios G, Blincoe K, Singer L, Germán DM, Damian DE (2016) An in-depth study of the promises and perils of mining GitHub. Empirical Software Engineering 21(5):2035–2071, DOI 10.1007/s10664-015-9393-5, URL https://doi.org/10.1007/s10664-015-9393-5
Lee T, Lee J, In HP (2015) Effect analysis of coding convention violations on readability of post-delivered code. IEICE Transactions 98-D(7):1286–1296, DOI 10.1587/transinf.2014EDP7327, URL https://doi.org/10.1587/transinf.2014EDP7327
Letouzey JL, Coq T (2010) 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, IEEE, pp 43–48
Likert R (1932) A technique for the measurement of attitudes. Archives of psychology
Maldonado EdS, Abdalkareem R, Shihab E, Serebrenik A (2017) An empirical study on the removal of self-admitted technical debt. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, pp 238–248
Martin RC (2009) Clean Code: A Handbook of Agile Sofware Craftsmanship. Prentice Hall
McCabe TJ (1976) A complexity measure. IEEE Transactions on Software Engineering SE-2(4):308–320, DOI 10.1109/TSE.1976.233837
McCall JA, Richards PK, Walters GF (1977) Factors in software quality: Vol. 1: Concepts and definitions of software quality. General Electric
Misra S, Akman I (2008) Comparative study of cognitive complexity measures. In: 2008 23rd International Symposium on Computer and Information Sciences, IEEE, pp 1–4
Munaiah N, Camilo F, Wigham W, Meneely A, Nagappan M (2017) Do bugs foreshadow vulnerabilities? an in-depth study of the chromium project. Empirical Software Engineering 22(3):1305–1347, DOI 10.1007/s10664-016-9447-3, URL https://doi.org/10.1007/s10664-016-9447-3

Murphy-Hill E, Black AP (2008) Refactoring tools: Fitness for purpose. IEEE software 25(5):38–44

Oram A, Wilson G (eds) (2007) Beautiful Code: Leading Programmers Explain How They Think. O'reilly

Palomba F, Tamburri DA, Fontana FA, Oliveto R, Zaidman A, Serebrenik A (2018) Beyond technical aspects: How do community smells influence the intensity of code smells? IEEE Transactions on Software Engineering

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

Piantadosi V, Fierro F, Scalabrino S, Serebrenik A, Oliveto R (2020) Replication package. https://github.com/stakelab/replication-readability-evolution

Posnett D, Hindle A, Devanbu P (2011) A simpler model of software readability. In: Proceedings of the 8th Working Conference on Mining Software Repositories, ACM, pp 73–82

Potdar A, Shihab E (2014) 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, IEEE Computer Society, pp 91–100, DOI 10.1109/ICSME.2014.31, URL https://doi.org/10.1109/ICSME.2014.31

Purushothaman R, Perry DE (2005) Toward understanding the rhetoric of small source code changes. IEEE Transactions on Software Engineering 31(6):511–526

Rajlich V, Gosavi P (2004) Incremental change in object-oriented programming. IEEE Softw 21(4):62–69, DOI 10.1109/MS.2004.17

Runeson P, Höst M, Rainer A, Regnell B (2012) Case Study Research in Software Engineering - Guidelines and Examples. Wiley, URL http://eu.wiley.com/WileyCDA/WileyTitle/productCd-1118104358.html

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

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

Scalabrino S, Linares-Vásquez M, Oliveto R, Poshyvanyk D (2018) A comprehensive model for code readability. Journal of Software: Evolution and Process 30(6), DOI 10.1002/smr.1958, URL https://doi.org/10.1002/smr.1958

Scalabrino S, Bavota G, Vendome C, Poshyvanyk D, Oliveto R (2019) Automatically assessing code understandability. IEEE Transactions on Software Engineering

Sheskin DJ (2007) Handbook of Parametric and Nonparametric Statistical Procedures, 4th edn. Chapman & Hall/CRC

Siegmund J, Kästner C, Liebig J, Apel S, Hanenberg S (2014) Measuring and modeling programming experience. Empirical Software Engineering 19(5):1299–1334, DOI 10.1007/s10664-013-9286-4, URL https://doi.org/10.1007/s10664-013-9286-4

Silva D, Tsantalis N, Valente MT (2016) Why we refactor? confessions of GitHub contributors. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM, pp 858–870

Spinellis D, Louridas P, Kechagia M (2016) The evolution of C programming practices: a study of the unix operating system 1973-2015. In: Dillon LK, Visser W, Williams L (eds) Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, ACM, pp 748–759

Thongmak M, Muenchaisri P (2011) Measuring understandability of aspect-oriented code. In: International Conference on Digital Information and Communication Technology and Its Applications, Springer, pp 43–54

Trockman A, Cates K, Mozina M, Nguyen T, Kästner C, Vasilescu B (2018) Automatically assessing code understandability reanalyzed: combined metrics matter. In: Proceedings of the 15th International Conference on Mining Software Repositories, ACM, pp 314–318

Tufano M, Palomba F, Bavota G, Oliveto R, Di Penta M, De Lucia A, Poshyvanyk D (2017) When and why your code starts to smell bad (and whether the smells go away). IEEE Transactions on Software Engineering 43(11):1063–1088

Vasilescu B, Serebrenik A, van den Brand MGJ (2011a) By no means: a study on aggregating software metrics. In: Concas G, Tempero ED, Zhang H, Penta MD (eds) Proceedings of the 2nd International Workshop on Emerging Trends in Software Metrics, WETSoM 2011, Waikiki, Honolulu, HI, USA, May 24, 2011, ACM, pp 23–26, DOI 10.1145/1985374.1985381, URL https://doi.org/10.1145/1985374.1985381

Vasilescu B, Serebrenik A, van den Brand MGJ (2011b) You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics. In: IEEE 27th International Conference on Software Maintenance, ICSM 2011, Williamsburg, VA, USA, September 25-30, 2011, IEEE Computer Society, pp 313–322, DOI 10.1109/ICSM.2011.6080798, URL https://doi.org/10.1109/ICSM.2011.6080798

Weyuker EJ (1988) Evaluating software complexity measures. IEEE transactions on Software Engineering 14(9):1357–1365

Wilcoxon F (1945) Individual comparisons by ranking methods. biometrics bulletin 1, 6 (1945), 80–83. URL http://www jstor org/stable/3001968

Zampetti F, Noiseux C, Antoniol G, Khomh F, Penta MD (2017) 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, IEEE Computer Society, pp 216–226, DOI 10.1109/ICSME.2017.44, URL https://doi.org/10.1109/ICSME.2017.44

Zampetti F, Serebrenik A, Penta MD (2018) Was self-admitted technical debt removal a real removal?: an in-depth perspective. In: Zaidman A, Kamei Y, Hill E (eds) Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018, ACM, pp 526–536, DOI 10.1145/3196398.3196423, URL https://doi.org/10.1145/3196398.3196423

Zhao Y, Serebrenik A, Zhou Y, Filkov V, Vasilescu B (2017) 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, IEEE Press, pp 60–71