



Software Source Code Readability

A Mapping Study

Andreas Bexell

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

The author declare that they are the sole author of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:

Author:

Andreas Bexell

E-mail: anbx17@student.bth.se

University advisor:

Dr Fabian Fagerholm

Department of Software Engineering

Faculty of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. Building software systems is an iterative and collaborative project, requiring developers not only to write code, but to maintain, expand, fix and enhance code already written. In order to do so, reading code is a central activity, and therefore it is important that code is written in a manner that makes it readable.

Objectives. To map the state-of-the-art of software source code readability and find the definitions and methods to measure it, and provide an overview of the kinds of factors considered to impact software source code readability, and to compare this to practitioners' experiences of software source code readability.

Methods. A systematic literature review of 76 studies in 72 papers from the last 40 years, explicitly concerning software source code readability, is compared with the results of five interviews with practitioners, of which three are case studies of commits explicitly targeting readability.

Results. While individual factors' contribution towards readability is studied with some success, more general modelling studies often suffer from methodological problems, making them difficult to apply in practice or in studies of the correlation between software source code readability and other metrics.

Conclusions. Key elements of the state-of-the-art have been implemented in practice, however, readability models are not used by the practitioners in this study. Several factors mentioned by practitioners are not considered by the studies included, and further qualitative study of software development practitioners may be needed.

Keywords: software architecture; software source code readability; systematic literature review

Contents

Abstract	i
1 Introduction	1
1.1 Why study readability at all?	1
1.2 Background	2
1.2.1 Software metrics	3
1.2.2 Static analysis	3
1.2.3 Readability of natural languages	3
1.3 Aim, objectives and thesis questions	4
1.4 Ethical, societal and sustainability aspects	5
2 Related Work	7
2.1 Software source code readability	7
2.2 Literature reviews	8
3 Method	9
3.1 Studies of software readability	9
3.1.1 Paper collection and selection	9
3.1.2 Data extraction	10
3.2 Developers' perspectives	10
3.2.1 Readability commits	11
3.2.2 Developer interviews	11
3.2.3 Extracting data from interviews	13
3.2.4 Validity of interviews	13
4 Results	15
4.1 Mapping readability	15
4.2 Modelling studies	17
4.3 Aspect studies	18
4.4 Theoretical studies	18
4.5 Correlation studies	21
4.6 Definitions	21
4.7 Taxonomy of factors	21
4.8 Empirical methodology	27
4.8.1 Collecting data from human subjects	27
4.8.2 Correlating metrics	29
4.8.3 Empirical methods summary	29
4.9 Developers' perspectives	30

4.9.1	Case Delta	30
4.9.2	Case Epsilon	31
4.9.3	Case Lambda	33
4.9.4	Interview Gamma	33
4.9.5	Interview Sigma	34
5	Discussion	37
5.1	Madness to the method	37
5.1.1	Artefacts of a flawed definition	37
5.1.2	Riding shotgun	37
5.1.3	Subject selection	39
5.2	Crumbling models	39
5.3	Developers' perspectives	40
5.4	Limitations and Threats to validity	42
5.4.1	Database selection	42
5.4.2	Search parameters	42
5.4.3	Software repositories	42
5.4.4	Qualitative methods, subject selection and transferability	43
5.4.5	Single researcher	43
6	Conclusions and Future Work	45
	List of tables	47
	List of figures	49
	References	51

Software development and software engineering are different from writing programs. While writing programs is often a solitary experience, development and maintenance of software systems are, at their cores, are collaborative experiences. Software systems can be vast (the Linux kernel is over 25 million lines of code) and long-lived, often living well beyond their originally intended life span, as was demonstrated by the Y2K problems.¹

Software architecture is often said to be the safeguarding of non-functional requirements. Key attributes of well-designed software systems are their maintainability and extendability. To maintain, fix and extend a software system efficiently and effectively, an important trait of that system's code is readability: if the software cannot be easily read and understood, it is hard to maintain, fix and extend. Such a software system is expensive and inefficient. Poor readability may also correlate to other quality problems (cf. Lee et al. [54, 53]), which may prove expensive.

On the other hand, formatting and style sets a precedence that may well outlive the functionality of a program (see Martin [59, p. 76]).

1.1 Why study readability at all?

When discussing the state of code readability research in the office, a colleague raised his eyebrows and said "I'm surprised that anyone would bother to study code readability, at all!"

There are, however, several good reasons to study the readability of software source code. Creating and maintaining software systems means working together with each others' code, and revisiting one's own code, to expand, enhance and correct it. This means that reading code is a skill all software developers must master, and that there is reason to study what properties of code make it more, or less, readable.

Software or code readability is studied from at least three perspectives:

The industrial perspective is concerned with the impact of readable code to software development professionals, and how that, in turn, affects other interesting aspects of software, such as quality, maintainability, and, ultimately, productivity. This

¹Y2K, simplified, denotes the problem that arose when programs created in the 20th century were constructed to hold years with only two digits, potentially making them fail at the turn of the century. That the system would indeed survive the turn of the century was often beyond the imagination of the original authors of those programs. (cf. Moffitt and Sandler [66])

perspective is in its core an economical one: the reasoning is that if industrial software developers spend much time reading code, and this can be made more efficient by increasing the readability of the code, then software development can be made faster and more cost-efficient (cf. Collar and Valerdi [22]). This perspective can be indicated for example by phrasing like “professional software developers read code for x % of their time”, typically in the introduction of a paper. (cf, for example Buse and Weimer [9], Aggarwal et al. [1], DeYoung and Kampen [27] among many other.)

The pedagogical perspective revolves around how to construct programming examples that are accessible to students, and in a readable way illustrate programming concepts in order to teach them effectively. This perspective is particular to programming education and is exemplified by Börstler et al. [12]. Studying readability from this perspective aims to create better and more efficient ways to teach programming concepts to students, and to better prepare students for a career in software development.

The cognitive perspective tries to advance knowledge about how our brains work when understanding code, and what kind of responses certain elements of source code trigger on a physical and cognitive level. This perspective is represented for example by Fakhoury et al. [32, 34, 33]. This line of study aims to further understanding of how the brain works to understand software and to understand the relations between cognitive load, effort, and understanding.

This list of perspectives, or reasons, to study readability, is not exhaustive, neither are the categories well-demarcated or mutually exclusive, but they offer an insight into what reasons there can be to study software readability as a concept and what the gains may be.

This master’s thesis endeavours to map the field of software source code readability research, in order to provide an overview of the state-of-the-art, but also to compare the state-of-the-art with the state of practice, by analysing commits from the industry specifically and explicitly addressing readability, and interviewing the authors of such commits, as well as other developers, about their attitudes and insights into the concept of readability, its meaning and significance.

1.2 Background

Even though readability of software have been a field of interests to academics, professionals and teachers since at least the 1970’s (see DeYoung [27]), and a SCOPUS search initially yields hundreds of articles mentioning readability of software, it remains that “the actual factors that make software easier to read or easier to comprehend are not well understood” (see Börstler and Paech [13]).

This section provides a summary of relevant concepts against which studies into software source code readability are conducted.

1.2.1 Software metrics

Since early in the development of programming and programming languages, there have been efforts to measure aspects of programs in an effort to determine factors such as performance and efficiency, but also quality and testability.

McCabe [62] formulated the concept of cyclomatic complexity of program code. This is a measure of the possible execution paths through a program, and is used as a measure of how many test cases are needed, but also sometimes as a measure of how complicated, understandable or readable program code is.

Halstead [40], anecdotally inspired by the general ideal gas equation, formulated several complexity measures based on the relations of operators and operands. This way, Halstead proposes formulas that relate to the V (volume), D (difficulty), and E (effort required to read/write) of a program.

Other commonly used metrics are, for example, lines of code, the ratio of blank lines and the ratio of comments.

Kitchenham [49] makes a mapping study of the state of software metrics, and shows that metrics are employed for a number of topics, including fault prediction, re-engineering, effort prediction.

Some of these metrics are employed as factors or studied as proxies or indicators of readability. (cf. Posnett et al. [75] and Börstler et al. [12])

1.2.2 Static analysis

Static analysis is the practice of automatically scrutinizing source code to find patterns of common problems. These problems can range from formatting mistakes to logical errors. Such analysis is often performed using a tool, that is sometimes integrated or interfaced from an integrated development environment (IDE).

Common such tools are *checkstyle*[8], focusing on coding style of Java code, *findbugs*[76], that checks Java bytecode for logical errors, and *PMD* [67], that for example finds unused variables and empty catch blocks in a number of languages.

It is theorised, for example by Lee et al., that poor readability of software can correlate with quality aspects indicated by such static analysis. [54, 53]

1.2.3 Readability of natural languages

Readability of natural languages is, compared to readability of software source code, relatively well defined.

In linguistics, there has been an effort to automatically rate the difficulty, grade level, or reading ease of natural languages. Specifically, in English, there have been several such constructs, including the Gunning fog index [39], the automated readability index (ARI) [87], the SMOG grading[63], the Coleman-Liau index [20] and the Flesch-Kincaid reading grade[48].

Such readability metrics, constructed for natural languages, are sometimes used to assess or evaluate the readability of source code (see Batool et al. [5], Pahal and Chillar [69] and Alawad et al. [2]), or inspire metrics that are meant to predict the readability of source code (see Börstler et al. [12]). It is not clear whether these readability metrics are directly applicable to code.

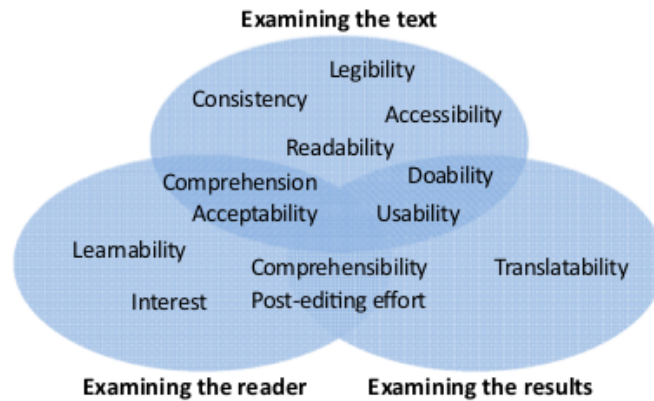


Figure 1.1: *Elements of texts, and their relations.* Venn diagram rendered by Börstler et al. [12, p. 235], citing Cadwell [15].

Readability and neighbouring concepts, as used in the studies of natural languages, have been elegantly described by Cadwell [15] (see figure 1.1).

1.3 Aim, objectives and thesis questions

The purpose of this study is to evaluate the state-of-the-art of software source code readability research, especially from the perspective of its usability and applicability to software development in the professional software development industry.

This study identify and categorizes prevalent definitions, methodologies and key factors of software readability research. It also critically cross-examines these, to provide an overview of the state-of-the-art in the field and identify key areas in which the research needs to be focused to further the understanding of software readability.

In order to achieve this, a systematic literature review is conducted, containing papers defining, measuring, and theorizing about software readability, and these elements are extracted, categorized, and cross-examined. This material is compared to developers' experiences, collected from interviews.

This will answer the following questions:

RQ1 What definitions of readability are used by the research community in papers explicitly concerning code readability?

RQ2 What methods are used to empirically measure code readability in papers explicitly concerning code readability?

RQ3 What is the validity of these methods as compared to the stated definition?

RQ4 What kinds of factors are considered, by the research community, in papers explicitly concerning code readability, to affect such readability?

RQ5 How do developers' experiences of software readability compare to researchers' factors of what is considered to improve code readability?

1.4 Ethical, societal and sustainability aspects

Unmaintainable code is often unsustainable code. Poorly understood code with poor readability leads to resource waste both in the form of bugs, but also because the software cannot be adapted, but needs to be rebuilt to adhere to new regulations, environments or requirements. To increase maintainability, by increasing readability, then, has an impact on the Sustainable Development Goals *8. Decent work and economic growth* and *9. Industry, innovation and infrastructure*, because maintainable software offers better potential for growth, and is a part of both innovation and infrastructure in a digitalized society, but it also affects goal *11. Responsible consumption and production*, because maintainable software is more resource-effective. [96]

This study contains interviews with human subjects. To protect the privacy of the interviewees, the results of the interviews are anonymized and presented in summarized form.

This chapter contains a brief overview of some related work into software source code readability. Since the thesis contains a systematic literature review of software source code readability studies, the bulk of such work is presented in the Results chapter 4.

Additionally, this chapter gives an overview of related systematic literature work that has inspired this thesis.

2.1 Software source code readability

Readability of source code has been studied to some extent. A SCOPUS search initially yields hundreds of papers on the topic.

Some papers endeavour to make a model to predict readability, usually by measuring the readability of code in some manner using human evaluation, comparing that to a set of factors, and using regression analysis, attaining a model based on the factors. This kind of work was pioneered by DeYoung and Kampen [27] and Jørgensen [47], gained traction with the influential work of Buse and Weimer [9, 10] and has been followed, for example, by Butler et al. [11], Posnett et al. [75], Dorn [28] and Scalabrino et al. [84, 82, 85]. Readability has also been studied empirically by Fakhoury et al. [32, 34, 33], who uses functional near-infrared brain imaging to measure cognitive effort.

Theoretical papers put forward arguments for a readability factor or propose a method for maintaining or enhancing readability, but do not back this up with any original empirical experiment. Early examples in the category include Elshoff and Marcotty [31], who discuss the merits of modularization, Crookes [24], emphasizing the importance of the right level of abstraction and the fit between the problem domain and the programming language, and Baecker [4], proposing the use of high-quality typography when printing code on paper. More recent examples include Sedano [86], who describes a method for code review meant to promote readability, Börstler et al. [14], who discuss perceptions of code quality, and Pascarella et al. [73], who proposes a taxonomy of comments.

Other papers evaluate a certain aspect or factor from a readability perspective. A typical research question in this category is “Is camelCase or snake_case more readable?”. Examples of such papers are Tenny [94], who investigates comments and how to organize procedures, Relf [77], who considers the importance of relevant naming, Binkely et al. [7] and Sharif and Maletic [88], who evaluate the relative merits of camel case and snake case. Further examples in this category include

Lanna and Amyot [52], Tashtoush et al. [93], Börstler and Paech [13], DosSantos and Gerosa [29], Wulff-Jensen et al. [101] and Wiese and Rafferty [100].

Yet other papers investigate the co-variance of a readability metric on one hand, and some other software metric on the other. This kind of paper has become common after the inception of the model created by Buse and Weimer [9, 10]. Early examples are authored by Lee et al. [54, 53], and this kind of papers are also authored by Pantiuchina et al. [71], Mannan et al. [57], Tashima et al. [92], Coleman [21] and Scalabrino et al. [83].

So far, to knowledge, only one attempt at a review of the field has been made, by Ribeiro et al. [79]. Only parts of the methodology are published, the rest needs to be retrieved from Ribeiro's master's thesis, written in Spanish [78]. They specifically pose the questions "Which attributes are used to evaluate the readability and comprehensibility of source code?"; "What are the measurement procedures for these attributes?" and "What are the existing relations between the identified attributes and the quality characteristics?". The questions and the selection criteria mean they review only papers fitting in the modelling category, as defined above, and they review 18 such papers. Key findings are that the models are contradictory, in that they assign the same attributes contradictory contributions towards readability.

2.2 Literature reviews

Kitchenham [49] makes a study to assess the state of software metrics. She examines 91 papers, and classifies them according to type as *theoretical*, *empirical* or *both*. She finds that many of the papers invalid empirical validations, make too many comparisons, and get contradictory results. She concludes that "software metrics researchers may need to refine their empirical methodology before they can answer useful empirical questions".

Verner et al. [98] make a tertiary study, a systematic literature review (*SLR*) of systematic literature reviews concerning *Global Software Development*. In this, they assess the quality of the studied SLRs using the York University DARE Criteria, and found that of the 24 studies, none reached more than three out of five possible points.

Kitchenham and Brereton [50] perform a systematic review of systematic review process research, and give a three-stage flow process for conducting SLRs, along with a list of commonly mentioned problems and advise. Common problems include that SLRs are time and effort consuming and that formulating research questions is hard, and common advice is to discuss the inclusion and exclusions of papers in a research group.

3.1 Studies of software readability

To map the field of readability of source code and software, this systematic literature study collects published papers on the topic, and these are scrutinized and categorized. The general method is heavily inspired by Kitchenham [49], who makes a mapping study of software metrics following three steps:

1. Identifying relevant papers
2. Data extraction, including classification of papers
3. Aggregation of results

3.1.1 Paper collection and selection

To find related relevant work, SCOPUS¹ was searched. In order to ensure the inclusion of all relevant studies, the search expression was designed to be wide. SCOPUS was searched on the 15th of April of 2020.

The fields *title* and *abstract* was searched, using the search string:

(software OR code) AND readability

This yielded 1620 documents. A manual assessment of those gave that many were concerned with automatic readability measures of natural languages of documentation, manuals or comments, and 104 documents were selected for further study based on title and abstract, on their relevance to software and program code readability.

Of the immediately rejected articles, some were medical, assessing the readability of information to patients or written communication between physicians. More software related, but nevertheless rejected, articles are about the readability of artefacts other than source code, for example a web page. Lately, studies of automatic readability of QR-codes have become common in the result.

The resulting filtered papers, now known to address software or code readability, were screened for the inclusion criteria:

contains at least one of

- a definition of readability
- an empirical measurement of readability
- a factor expected to impact readability.

¹<https://www.scopus.com>

The SCOPUS results, with these criteria applied, yielded 74 papers.

In order to mitigate the fact that only one search engine was used for the initial search, the reference lists of the papers now included were examined, and again papers fulfilling the inclusion criteria were added to the study corpus. This way, another three papers were included.

One exclusion criterion was used: if a paper found in SCOPUS, or in the references lists of included papers, was not accessible through the BTH library, on google scholar or from a wider web search, it was excluded. In the end, no papers were excluded based on this criterion.

No quality criteria were applied to the studied papers. A part of mapping the field of software source code readability is determining the quality of the studies conducted.

In total, 76 studies from 72 papers explicitly concerning readability of software source code are selected for inclusion in this study.

3.1.2 Data extraction

From the included papers, three kinds of information are specifically extracted, by a thorough manual reading of the papers:

Any explicit *definition* of readability. Such definitions are often, but not always, indicated by “readability is ...” or “readability is defined as...”.

Any *factors* taken to impact readability are noted. This can be things as proper layout or the presence of comments.

For *empirical* papers, the *method* of the underlying human experiment is examined. Specifically, the measurands, the subject selection and the sizes of the studies are noted.

This data will allow answering RQ1-4. (See section 1.3, p. 4)

3.2 Developers’ perspectives

Since the explicit perspective of this thesis is that of the software industry, the state-of-the-art, as represented by RQ1-4 are completed by RQ5 (see section 1.3, p. 4).

In order to ground the research communities’ perspectives in an industrial outlook, an effort is made to collect the perspective of professional developers. Two sets of ontological properties are sought for in this study: one is the *perspectives, experiences and accounts* of professional software developers, but also the *understandings, interpretations and knowledge* such professional developers can bring. (cf. Mason [60]) This is done by interviewing professional software developers about readability of software source code in general, and in context of a commit specifically targeting readability.

The purpose of the interviews is not to uncover generalisable truths – rather it is to broaden the perspective and give an indication about the level of agreement or disagreement between how researchers and practitioners discuss and think about readability of software source code, for example in areas such as terminology, concepts, perceptions and experiences, and to give an indication about the completeness of the state-of-the-art.

3.2.1 Readability commits

Mining software repositories for commits specifically targeting the readability of the source code is an established process, however, the analysis of these commits are mostly quantitative (see for example Pantuchina et al. [71] and Fakhoury et al. [33]) or correlative.

The readability commits are found using `git` (see listing 3.1).

```
# git log --all -i --grep reada
```

Listing 3.1: *The git log of all branches is filtered for commits with messages containing “reada”, case insensitive. This matches both “readability” and “readable”, for example, and the result needs to be manually curated.*

The purpose is to examine commits explicitly about *readability*. Several other kinds of changes, especially refactorings, can affect readability (see Silva et al. [90]), but for this study only commits in which the developer has explicitly intended to change readability are considered, in order to capture what developers mean when they think specifically about *readability*.

This study takes a qualitative approach to readability commits. Software repositories are chosen and mined for readability commits. These are taken as examples and starting points in interviews with involved developers and become a focal point for a discussion about readability. In addition, these commits then provide a common reference point to enhance *interpretative* validity. (cf. Maxwell [61, p. 9f])

Readability commits were sought for in eight different repositories with active developers from a major international consumer electronics company. Six of the repositories constitute proprietary Android system applications, and are mainly written in Java. One repository is a test harness, mainly written in Python 3, and one is a test framework, mainly written in JavaScript. These particular repositories were selected based on their availability, and the availability of their respective active developers, to the author.

3.2.2 Developer interviews

When possible, the developers involved in the creation readability commits, either as authors or reviewers, found in the software repository mining described above, are contacted and interviewed about their general attitudes and viewpoints on readability, as well as their specific intentions and the readability problems addressed in their specific commits, and how their commit improved readability.

Since commits with a commit message specifically mentioning readability are rare, these interviews are complemented by interviews with one senior developer, and one relatively junior one. This follows the principle from Mason of selecting to “gain access to relevant data sources” and selecting “for the purposes of gaining meaningful insights”. [60, p. 53]

The method of interviewing is what Mason refers to as “qualitative interviewing”: the interviews are in-depth and semi-structured. [60, p. 109] A simple interview guide was devised to keep the discussions comparable and on track, but the guide as a whole does not constitute an open-ended questionnaire, rather the questions should be understood as “starting points for discussion” or *themes*. [60, p. 110] Such

interviews allows for subjects to reason freely around the topic, and has the benefit that they can uncover aspects, perceptions and reasoning not previously considered by the researcher.

1. What is your experience with software? How many years have you been a software professional, and in what contexts?
2. What is readability to you? What do you associate with “readability” in the context of software source code?
3. In this specific commit – how does the change increase the readability?
4. Which static analysis tools do you know of, and which do you regularly use?
5. Which automated software readability metrics do you know of, and which such metrics do you regularly use?

The subjects were informed that the interviews were for the purpose of research into software source code readability and were to provide an industrial practitioners’ perspective into such research. The questions are designed to be open, as to not lead the interviewees, and to be conversational, as to not intimidate subjects. The purpose of the questions is not so much to get specific answers, but rather to serve as a road-map to keep a discussion about readability of software on track.

The order of the questions is significant: the first is a warm-up biographical question, the second and third allow the interviewees to reason freely, and require relative ease of mind, while the fourth and fifth, which may seem like a test or a basis of evaluation, are placed last, in order for them not to interfere with the ease of mind needed for questions two and three. The last one, where it is expected that many developers will not know an answer, may evoke a sense of failure, is placed last as to not let this feeling impair or distract the answers to the other questions.

Question 3 is omitted from the interviews, in which there are no specific commits to discuss.

Questions 4 and 5 are questionnaire-like, and serve to establish the prevalence of automated tools use in general and concerning readability in particular.

The interviews were conducted over telephone, audio only, and one by one. The interviews were not recorded, but notes were taken during the interviews to ensure the descriptive validity (cf. Maxwell[61, p. 7ff]), and any uncertainties were continuously clarified during the interviews to ensure interpretative validity (cf. Maxwell[61, p. 9ff]).

In total, five interviews were conducted. They lasted between 20 and 120 minutes. All subjects are native Swedish speakers, and so the interviews were conducted in Swedish, although technical terms were frequently used in English. Three interview subjects were selected based on their involvement in a readability commit (see section 3.2.1), and two based on their experience: one developer with long experience, and one with relatively short experience of professional software development. (cf. Maxwell citing Patton on “purposeful” selection [61, p. 13] and Mason [60, p. 53].)

3.2.3 Extracting data from interviews

While extracting data from qualitative interviews can be a laborious task (cf. Mason [60, p. 187]), this study benefit from the context at hand. The purpose of the interviews is to extract data about professional software developers knowledge and perspectives on what constitutes software source code readability and how it can be understood – that is, how software source code readability is understood – and about the practitioners' perspective and knowledge about the factors that affect such readability, as it is understood. This is to be compared to the definitions and factors considered by the state-of-the-art.

This means that the extraction of data from the interviews is relatively easy: it is a thematic analysis where at least a subset of the themes are already provided by the analysis of the state-of-the-art, and as such it is about listening for markers for ontological and epistemological statements about readability – what constitutes readability and how it is perceived to come into existence; and to capture those statements as clues to definitions or factors affecting readability.

3.2.4 Validity of interviews

Maxwell proposes a typology of validity in qualitative research in five dimensions: *descriptive*, *interpretative*, *theoretical*, *evaluative* validity and *generalizability*. [61]

The *descriptive* validity is ensured by taking close notes of the interviews. These notes are examined post facto. To ensure the *interpretative* validity, it is sometimes necessary to, during the interviews, ask for clarifications. This way it can be made reasonably sure that the subject and the interviewer has a common understanding, and this is a vital part of the interview process, especially since one of the purposes of the interviews is to determine the generalizability of the terminology used in the state-of-the-art. It is therefore not enough to accept a term used in the state-of-the-art during an interview without also asking what the subject refer to using that term.

Theoretical validity of statements made in the interview balances on the limit of the scope of this thesis. As the purpose of the interviews is to assess the completeness of the state-of-the-art, there is no need to evaluate the validity of any new factors or definitions uncovered in the interviews, merely to determine whether they warrant further research or not. This also means that this thesis makes no claims on *evaluative* validity – such is left for future research.

The *generalizability* of the findings of the interviews should not be understood in the same manner as for quantitative or experimental research. The definitions or factors mentioned in interviews will not be the same in a repeated study – however if interviews give an indication of incompleteness or terminology confusion in the state-of-the-art, it can be considered likely that repeated studies will continue to indicate such incompleteness.

4.1 Mapping readability

Upon review it becomes clear that the included studies are qualitatively different: there are different *kinds* of readability studies. To better study and compare them, they need to be categorized. See figure 4.1.

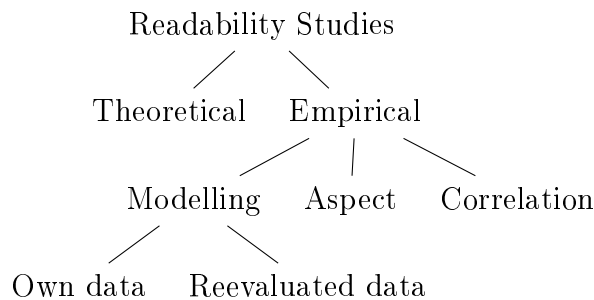


Figure 4.1: *An hierarchical view of the categories of studies described in the papers included in the mapping study.*

This first divider is the one between theoretical studies and empirical ones. Theoretical studies, in this context, offer a theory on readability or factors, but do not anchor this theory by measurements or collected data. These studies can have other empirical elements: a common theme is a tool to automatically detect the presence of the theorized factors in code, but they do not empirically validate the correctness of the theorized readability factors. Empirical studies, on the other hand, do analyse some kind of measured elements of readability to explore or draw conclusions about readability.

These empirical studies, in turn, can be divided into three main themes: Studies using data collected from an experiment on humans to discern the impact on readability from one or a few factors are called *Aspect* studies throughout this master's thesis. A typical question posed in aspect papers is "Is camelCase or snake_case more readable?". Another family of similar studies take a wider grip: they analyse data from an experiment on humans and perform a multiple correlation analysis towards a given set of factors, in order to build a generic model meant to predict readability. These studies are labelled *modelling* studies throughout this master's thesis. Finally, a family of papers employs a readability metric on code along with some other kind of measure, to draw conclusions from their correlation. A typical question posed by

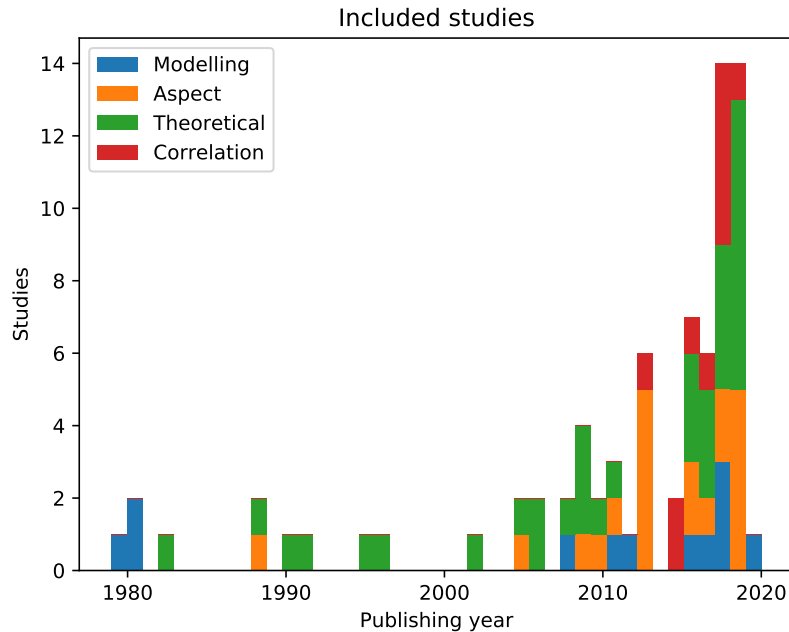


Figure 4.2: *Timeline of publishing years of studies in papers, colour coded by category. Note that one publication can contain more than one study, and that such studies can be of the same or of different categories.*

these studies is “Is less readable code more bug-prone?”. Throughout this master’s theses, this family of studies is labelled *correlation* studies. The studies are presented according to their respective category.

This classification is qualitative, in that it divides the studies into *kinds*, and hierarchical, in that the *Modelling*, *Aspect* and *Correlation* studies all are subcategories of the root category *Empirical* studies. (cf. Usman et al. [97, sec. 2.4-2.5]) The classification is based on a bottom-up perspective.

Figure 4.2 illustrates the publication times and kinds of the included studies explicitly of readability of software source code, to date. It is apparent that while the field has been studied for more than 40 years, it has gained considerable traction since the late noughties. Especially correlation studies have appeared quite recently – most probably because the models published around 2010 have become widely known and are easy to work with, therefore also to correlate against other metrics.

The empirical studies come in three categories: *modelling* studies endeavours to build a mathematical model from factors as compared to empirical data in order to be able to mathematically predict the level of readability of code. Some of these studies create and evaluate their own original data set, while others reuse data from previous studies in order to evaluate another model containing other factors. *Aspect* studies investigate the readability impact of typically one or two factors. *Correlation* studies investigate the covariance between readability and some other aspect of software quality. Typically, these papers use a model to predict readability, and an automated tool to compare the measured readability with some other software metric.

4.2 Modelling studies

Modelling studies try to find a general mathematical model in order to predict readability. To do so, they collect and reuse data from human annotators or judges, and compare this data to a set of factors presumed to impact readability. Through multiple regression, each factor can be assigned a significance, and a weight, depending on its positive or negative contribution towards readability, as perceived or experienced by the humans.

First author	Year	Data	Assessment	Comprehension	Effort	Subjects
DeYoung [27]	1979	Own	Likert			31 Students
Jorgensen [47]	1980	Own	Likert			10 Devs
		Own	Likert	Questions		100 Students
Buse [9, 10]	2008	Own	Likert			120 Students
Posnett [75]	2011	[9]	Likert			120 Students
Dorn [28]	2012	Own	Likert			3668 Students, 1800 Devs
Scalabrino [84]	2016	[9], [28], Own	Likert			3818 Students, 1800 Devs
Scalabrino [82]	2017	Own	Likert	Questions	Time	41 Students, 5 Devs
Mi [65]	2018	[9], [28], [84]	Likert			3818 Students, 1800 Devs
Scalabrino [85]	2018	[9], [28], [84]	Likert			3818 Students, 1800 Devs
Choi [19]	2018	Own	Likert			N/A
Choi [18]	2020	Own	Likert			45 Devs

Table 4.1: *An overview of modelling studies. For each study, the origin of the data is reported. Also, for each study, the methods of measuring assessment or impression, comprehension and effort are reported, alongside the subjects contained in the study, with Students denoting people enrolled in university studies, while Devs denote professional software developers.*

Table 4.1 provides an overview of the modelling studies in the included papers. It is apparent that these studies accumulate data and grow over time, and that the study done by Buse and Weimer in 2008 [9] and republished in 2010 [10] was influential and has inspired more efforts. The data set has been reanalysed and expanded several times by several authors.

However, it is also apparent that the data accumulated that lies the foundation

of the modelling studies is mainly based on an assessment of readability, rather than measuring effort or comprehension. Of the 12 studies, only two (Jorgensen [47] and Scalabrino et al. [82]) measure comprehension, and only one measure any proxy for effort (Scalabrino et al. [82] measure time). This means that the data the models are based upon to a large extent measure only an impression of readability, which may or may not be representative of readability as such, depending on the definition of the concept – see also sections 4.6 and 5.1.1. In fact, Scalabrino et al. conclude that readability, as modelled, does not correlate significantly with understanding [82, p. 425], indicating that impression of “ease of understanding” may not be the same thing as actual “ease of understanding”.

4.3 Aspect studies

Aspect studies try to determine the readability impact of certain factors. The number of factors considered varies between the studies: while some evaluate only one factor, some take over 20 into account. As compared to the modelling studies (see Section 4.2), there is a higher tendency to validate comprehension and measure a proxy for effort in the subjects. This may indicate that the aspect studies work with a definition of readability different from the one most prevalent in the modelling studies. (See also 4.6)

Table 4.2 provides an overview of the aspect studies included in this mapping study.

Binkley et al., on one hand, and Sharif and Maletic on the other, study the same aspect: are identifiers using camel case (new words in multi-word identifiers are marked by a capital letter, like so: *camelCase*) or using snake case (where words in identifiers are separated by an underscore, like so: *snake_case*) more readable? They use similar methods and yet reach diametrical results. Both fail to observe that while Binkley et al. partly use subjects trained in, and explicitly preferring camel casing [7, p. 163], Sharif and Maletic use subjects “historically trained mostly in the underscore identifier style” [88, p. 200]. It cannot be ruled out that training and preference of the subjects are more influential in the results, than the studied difference in identifier naming style.

4.4 Theoretical studies

Theoretical studies (see table 4.3) offer definitions of readability, or factors that may affect readability, but do not empirically evaluate these factors. This does not generally mean that those papers cannot have some other empirical element, not directly studying readability.

Some of these papers, interestingly, propose models or metrics of readability, without an empirical evaluation: Aggarwal et al. [1], Chhabra [17], Hourani et al. [42] and Siddiqui and Ahmand [89].

A recurring theme is naming, mentioned by Deißeböck and Pizka [25, 26], Høstand Østfold [43], Butler et al. [11], Holzmann [41], and Liu et al. [55]. This is a topic that also has been evaluated in the aspect studies (see table 4.2).

First author	Year	Aspect(s) studied	Method	Effort	Comprehension	Subjects
Tenny [94]	1988	Code locality, comments	Questions	Time	Correction	148 Students
Relf [77]	2005	Assisted Naming	Tasks		Correction	69 Students, 10 Devs
Binkley [7]	2009	CamelCase snake_case	v. Pick same	Time	Correction	135 Students
Sharif [88]	2010	CamelCase snake_case	v. Pick same	Time, Eye tracking	Correction	15 Students
Lanna [52]	2011	Colour highlighting	Tasks	Time	Correction	13 Students, 3 Devs
Tashtoush [93]	2013	23 programming features	Ranking of features			141 Devs
Feigenspan [35]	2013	Colour highlighting	Tasks	Time	Correction	52 Students
		Colour highlighting	Tasks	Time	Correction	10 Students
		Colour highlighting	Tasks	Time	Correction	10 Students
		Colour highlighting	Tasks	Time	Correction	14 Students
Batool [5]	2015	Comparison of C# and Java over 22 features	Questionnaire			15 Devs
Marter [58]	2016	Naming	Reading	Time	Correction	32 ppl
Börstler [13]	2016	Method chains and comments	Likert, Cloze	Time	Correction	104 Students
Ribeiro [79]	2017	Indentation, naming, size	Likert			38 Students
Fakhoury [32]	2018	Linguistic anti patterns	Bug finding	Time, Cog. load	Correction	70 Students
DosSantos [29]	2018	11 practices: naming, structuring, nesting	Likert			55 Students, 7 Devs
Wulff-Jensen [101]	2019	Indentation, naming, comments	Likert	Eye tracking	Correction	21 Students
Lucas [56]	2019	Lambda expressions	Likert			28 Students
Johnson [46]	2019	Nesting, do-while	Review	Time	Correction	241 Students, 61 Devs
Wiese [100]	2019	“Novice patterns”	Review		Correction	231 Students
Fakhoury [34]	2019	linguistic and formatting inconsistencies	Bug finding	Time, Cog. load	Correction	25 Students

Table 4.2: An overview of studies evaluating certain aspects’ impact on readability.

First author	Year	Topic
Elshoff [31]	1982	13 good practices in code design
Baecker [4]	1988	Typographical presentation
Sametinger [81]	1990	A tool for browsing C++ Software
Smith [91]	1991	A teaching method
Jaaksi [44]	1995	A design method
Parrish [72]	1996	A design methodology
Aggarwal [1]	2002	A model proposition
Deißenböck [25, 26]	2005	Concise and consistent naming
Collar [22]	2006	Role of software readability on development cost
Kussmaul [51]	2008	Proposes a new syntax
Høst [43]	2009	Assisted naming
Palmer [70]	2009	Proposes a new syntax
Butler [11]	2009	Identifier naming
Beyer [6]	2010	Dependency degree
Chhabra [17]	2011	A cognitive complexity measure
Silva [90]	2016	Analysis of refactoring
Sedano [86]	2016	A review method
Holzmann [41]	2016	Identifier naming
Newman [68]	2017	Syntactic restructuring
Ribeiro [79]	2017	Factor review of 18 empirical papers
Pahal [69]	2017	A listing or readability measures
Mi [64]	2018	Gamification of writing for readability
Borstler [14]	2018	Interviews with students, developers and educators
Chaweewan [16]	2018	Aptitude evaluation based on B&W [9]
Pantiuchina [71]	2018	Compares readability commits with metrics
Rodrigues [80]	2019	Specification of coding conventions
Pascarella [73]	2019	Classification of comments
Hourani [42]	2019	Proposes a new metric
Liu [55]	2019	Identifier naming
Fakhoury [33]	2019	Compares readability commits with metrics
Siddiqui [89]	2019	Proposes a new metric
Aragão [3]	2019	Developer profiling
Jiang [45]	2019	Evaluation of a tool

Table 4.3: *The included theoretical studies, along with their respective topics.*

Outliers in this category are Pantiuchina et al. [71] and Fakhoury et al. [33], that both mine software repositories for commits meant to affect readability, and compares the results with metrics. As such, they are candidates also for the category *correlation studies* (see section 4.5), however, they remain in the theoretical studies category simply because they do not, as the correlation studies do, compare a readability metric with something else. Interesting to note about these studies is that both find that readability metrics (Buse and Weimer [9], Dorn [28] and Scalabrino et al. [85]) fail to reliably capture the developers' intended readability improvements.

4.5 Correlation studies

The correlation studies have become popular since the inception of automated readability models, and generally employ an automated readability metric, and compare the output of this metric to some other metric or automated result. An overview of the included correlation studies is presented in table 4.4

An outlier in this category is Börstler et al. [12], that proposes a new model and could be a modelling paper. However, the proposed model is not evaluated against raw data, that is, not towards observations of humans, but rather in the ways it *correlate* to other metrics (some of those being other readability metrics), and so is a study more of a correlation than a modelling study in itself, by the categorization employed in this thesis.

Interesting is Scalabrino et al. [83], which essentially correlate metrics of perceived readability (see sections 4.2 and 5.2 for further discussion) with comprehension and time, and find that they correlate badly.

4.6 Definitions

Of the 71 papers, 22 use an explicit definition. 13 make their own definition, one paper makes use of a definition from outside the field, making a total of 14 explicit definitions. See table 4.5.

The most prolific and influential definition was coined by Buse and Weimer in [9] and is used in eighth of the other articles.

Answer to RQ1: Three concepts are recurring in the definitions, in different combinations. Of the 14 definitions, eight connect *readability* with *understanding* or *comprehension*, while one mentions *access the information*, which can be interpreted as a circumlocution for understanding. Ten explicitly mention *ease* or *effort*. Seven of the definitions explicitly mention ease or effort with regards to understanding. Five definitions define readability as a subjective judgement or impression, and all of these also mention *effort* and *comprehension*. See also tables 4.5 and 4.6.

4.7 Taxonomy of factors

Upon close inspection of the papers, it becomes clear that the sheer number of factors considered, in the papers, to impact readability is hard to grasp. Between papers, it

First author	Year	Readability Metric	Correlates to
Lee [54]	2013	B&W [9]	Checkstyle & PMD violations
Lee [53]	2015	Posnett [75]	Checkstyle violations
Börstler [12]	2016	SRES	B&W[10], Posnett [75], CC [62], E [40], MI [99], CC/NOS NOS, LOC, CD
Fernandes [36]	2017	CheckIO votes	SLOC, CC [62], Comments
Mannan [57]	2018	Posnett [75]	inFusion
Tashima [92]	2018	Levenshtein distance of identifier names	Bugs
Grano [38]	2018	B&W[10], Posnett[75], Scalabrino [84]	Automatic tests
Alawad [2]	2018	ARI [87], SMOG [63], FKI[48], GFI[39], CLI[20], B&W[9]	Halstead V[40], CC[62]
Coleman [21]	2018	B&W [10], Posnett [75], SRES [12]	Beautification
Scalabrino [83]	2019	B&W [10], Posnett [75], Dorn [28], Scalabrino [84]	Understanding, time

Table 4.4: Overview of studies focusing on the correlation between readability measures on one hand, and some other metric or measure on the other. *B&W* denotes the readability metric devised by Buse and Weimer [9, 10], *CC* is McCabe’s Cyclomatic Complexity [62], *E* and *V* are Halstead’s effort and volume, respectively [40], *MI* is maintainability index by Welker et al. [99], *NOS* is Number of Statements, *LOC* is Lines of Code, *CD* is comment density, *SLOC* is Source Lines of Code. *inFusion* is a static analysis tool detecting “code smells”-

Definition	Origin	Used by
“ease of comprehension”	[22]	
“a human judgment of how easy a text is to understand”	[9]	[10, 11, 93, 54, 53, 57, 71]
“a subjective impression that programmers have of the difficulty of code, as they try to understand it”	[75]	
“a property that manipulates how easily a given piece of code can be read and understood”	[5]	
“what makes some texts easier to read than other”	[30]	[12, 13]
“a quality characteristic based on the human judgment of how easy is to read the written code”	[79]	
“the proficiency of software code which makes it legible and comprehensible even for a non-technical staff”	[69]	
“Readability is about understanding code”	[21]	
“the extent of how easily the program code can be read”	[19]	
“a human judgment as to how much source code is understandable and easy to read”	[2]	
“the effort of the developer to access the information contained in the code”	[85]	
“a human judgment of how easily a program’s source code can be read and understood”	[65]	
“the aesthetics and general concerns related with code legibility”	[3]	
“readability is a self-reported opinion, while comprehension is objective”	[100]	

Table 4.5: *Definitions of readability, their origins and usage in included papers. Note that DuBay [30] is not an included paper, but a definition from outside the field of computer science.*

	Impression		Comprehension		Effort	
	Def	Pap	Def	Pap	Def	Pap
Impression	5	12	5	12	5	12
Comprehension			9	16	7	14
Effort					10	19

Table 4.6: *Concepts connected to readability, and their inclusion and combinations in the 14 definitions and in the 22 included papers employing an explicit definition. All definitions mentioning Impression also mention Comprehension and Effort.*

Category	Examples of Factors
Language	naming, vocabulary
Layout	indentation, blank lines, spaces, line breaks
Presentation	colour highlighting, fonts
Size	measures of the size of the code
Code	having specifically to do with code execution
Scope	Having to do with scope of identifiers
Context	Comments, documentation
Entropy	entropy measures

Table 4.7: *The categories of factors found in the included studies.*

varies from studies considering one or two factors or aspect, to studies considering more than 120 factors. In order to get an overview of the factors, it is thus necessary to group them in a comprehensible way.

Implicitly, some efforts have been done in this direction, but those are unstructured and lack clear demarcations:

Scalabrino et al. [84] write in their abstract that at that point “software readability models take into account only structural aspects of source code, *e.g.*, line length and a number of comments” and explicitly aim to add “textual aspects of source code encapsulated in identifiers and comments”. However, when Börstler et al. write about structure, they mean “well structured, modular, cohesion, low coupling, no duplication, decomposition” [14, p. 75].

For this thesis, a qualitative, enumerative taxonomy is developed from a bottom-up perspective. (Compare with nomenclature used Usman et al. [97, p. 44].) This means that the categories cover the different lines of inquiry the included papers take to be affecting or indicative of readability, but may not cover factors not considered in the included papers. For an overview of the categories, see table 4.7.

Language factors include naming, as studied, for example, by Butler et al. [11], Holzmann [41], Liu et al. [55] and Marter et al. [58], but also what Scalabrino et al. call *textual features* [84] and the way the texts are written, as studied by Binkley

et al. [7] and Sharif and Maletic [88]. Another example of language factors are the *linguistic inconsistencies* studied by Fakhoury et al. [34].

Layout factors concern indentation, blank lines, and spaces, for example. This is sometimes referred to as *structural* features (cf. Scalabrino et al. [84]), but *structure* in the context of software development is ambiguous, as it can be taken to refer to the design of a system (see interviews with *Gamma* 4.9.4 and *Sigma* 4.9.5 and Börstler et al. [14]). Therefore, the term *layout* is chosen in this thesis' taxonomy. Layout factors are commonly controlled with coding style guidelines. Layout features are included for example in the model devised by Buse and Weimer [9, 10], and studied by Lee et al. [54, 53]. Note that some layout features may be mandated by some languages: for example, while indentation of blocks is not required for example in C or Java, such indentation has a syntactic meaning and is required in Python and make.

Presentation factors are introduced by the tools used to present the code, rather than of the code itself – that is, these factors are typically not saved with the code. This category contains syntax highlighting, other forms of colour coding (see Feigenspan et al. [35]), use of typography (see Baecker [4], and even making code hypertext, to be able to follow links to declarations, for example (see Sametinger [81]).

Size factors capture different aspects of size of the code. Number of Statements and Lines of Code as used by Börstler and Paech [13] and Fernandes et al. [36], for example, are in this category.

Code factors are factors having specifically to do with the fact that the examined text is executed: it makes assignments and it loops, for example. McCabe's cyclomatic complexity [62] and Halstead's complexity measures [40] are both examples of factors in this category. Some of the measures and factors may not have perfect orthogonality to size measures: for example, a higher Halstead volume probably correlates to higher size measures. Still, a Halstead V capture properties of complexity and indicators of cognitive load in a way that size measures alone do not.

Scope has to do with the question *For how long do I need to keep this identifier committed to memory?* It is studied, for example, by Chhabra [17] and Tashtoush et al. [93], and mentioned in the interview with *Lambda* (see section 4.9.3).

Context factors are factors that provide additional information, besides the code and may or may not be collocated with the code. Comments and documentation are both examples of contextual factors. Contextual factors are included, for example, in DeYoung and Kampen [27], Jørgensen [47], Tenny [94], and in depth by Pascarella et al. [73]. Such features are also mentioned in the interview with *Sigma* 4.9.5.

Factor category	Model	Aspect	Theoretical	Correlation	Sum
Language	9	11	10	5	35
Layout	8	7	4	2	21
Presentation		5	2		7
Size	5	2	2	1	10
Code	9	9	8	5	31
Scope		1	1		2
Context	6	4	7	4	21
Entropy	1				1

Table 4.8: Number of studies concerned with a certain category of factors, separated by the types of studies and in total. Note that one study can concern several different categories of factors.

Entropy factors are a measure of disorder or uncertainty in a system. Posnett et al. [75] calculate the Shannon entropy over bytes and terms for their model of readability.

Worth noting is that not all of these factors would be considered a part of *readability* in a traditional sense of the word: natural language studies of readability do not typically associate presentation or layout with readability, but rather with legibility (cf. DuBay [30], Cadwell [15] and figure 1.1, p. 4), and, of course, natural language texts rarely branch conditionally, as code execution tends to do, adding a layer of complexity. Since they are, however, studied as factors of readability in the context of papers explicitly concerned with the readability of software source code, though, in this context it is appropriate to call them categories of factors of readability.

Table 4.8, page 26 provides an overview of to what extent the different categories of factors are studied in the different kinds of papers. From here, it is apparent that *Language* and *Code* factors are of major concern to the research community: in total, nearly half of the 76 included studies use some kind of factor from those categories.

Layout factors, containing, for example, tabulation and new lines; and *Context* factors, containing, for example, comments; are both studied in 21 studies, making them the third most common categories of factors studied.

Interestingly, *Size* factors are considered in only 10 studies. Intuitively, it would seem size is an important indicator of readability: it may seem too trivial for aspect and theoretical studies. Then again, only five of the 13 modelling studies expect the size to be indicative of readability. To some extent, some size measures may be derived from some measures in the *Code* category: for example, a high Halstead volume may be indicative of a large size, by some measure, and *Code* factors are more common in the modelling studies. Still, it would seem easy to include *Size* factors as well.

Presentation factors are only considered in just under 10% of the studies. This is not surprising, as it is on the fringe of the definition space. *Presentation* factors

are not properties of the code itself, but rather of the tools used to display the code, and so cannot be used in modelling studies, studying the code itself, independently of a human reader. Arguably, *Presentation* factors are more in the realm of legibility than readability. This is not to say, however, that good typography, as studied by Baecker [4], or colour coding, as studied by Lanna and Amyot [52] and Feigenspan et al. [35] cannot decrease the effort to understand certain aspects of code.

Scope factors take into account how much a reader needs to keep committed to memory when reading the code. This is discussed by Chhabra [17] and Tashtoush et al. [93], and also mentioned in the interview with *Lambda* (see section 4.9.3). This is a kind of factor that may warrant further study.

Including a measure of *Entropy* factors are an innovation by Posnett et al. [75] for their remodelling of the data provided by Buse and Weimer [9, 10]. *Entropy* factors have not been considered in any of the other included papers, and it would be interesting to evaluate the impact of character and token entropy on readers' effort and cognitive load.

Answer to RQ4: The most commonly considered kinds of factors are concerned with *Language* and *Code*, followed by factors concerning *Layout*. Less common are factors concerning *Size* and *Presentation*. Factors concerning *Scope* are studied in two papers, and *Entropy* is used as a factor in one paper. See also table 4.8.

4.8 Empirical methodology

Mainly two types of empiric methods are used in the papers included in this study. *Aspect* studies and *Model* studies both collect data from human subjects or judges, while the *Correlation* studies correlate two or more metrics to determine their relationship.

4.8.1 Collecting data from human subjects

32 studies in the *aspect* and *modelling* categories use data collected from human subjects. This is done by creating or assembling a number of example programs or snippets of code, presenting these to human subjects, and perform some kind of measurement of the humans' impressions or other reactions to those programs.

Some of the studies employ student subjects, while others use software professionals as subjects. Yet others employ different mixes of student and professional subjects. The sizes of the studies vary by several orders of magnitude. Figure 4.3 provides an overview of the sizes and mixes of the included studies.

When conducting data collection from human subjects, there are three basic measures employed by the research community:

Impression of readability, is usually measured on a Likert scale, and measures the opinion or initial assessment of perceived readability. This is done on relatively short snippets of code. 17 of the studies employ some kind of measurement of the impression of readability.

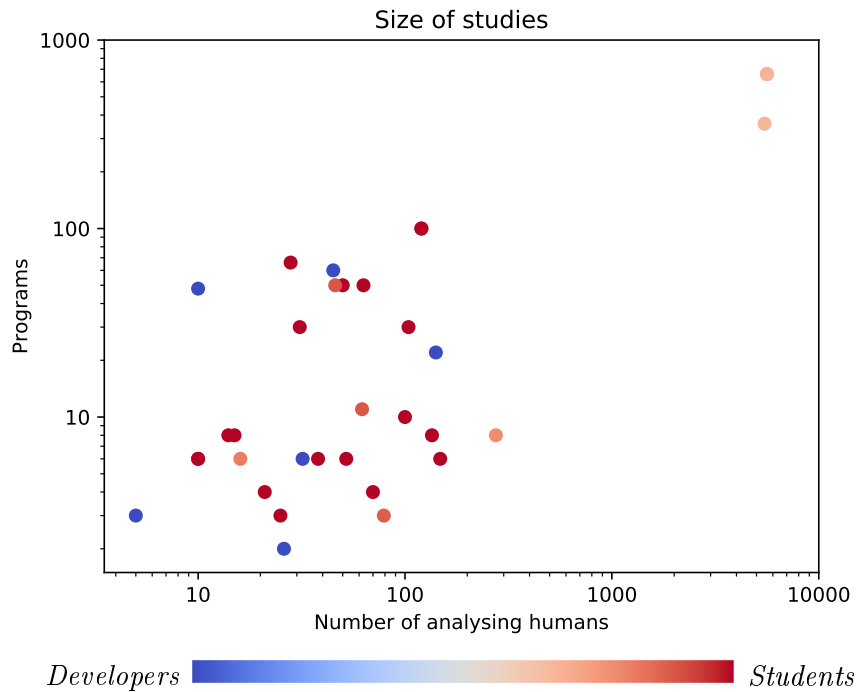


Figure 4.3: Sizes of empirical studies, with regards to the number of programs analysed, the number of human judges and the kind of judges. Professional developers colour the dots towards blue, while students colour the dots towards red, along the scale. Note the log-log-scaling. Also note that especially for some of the larger studies, all judges do not analyse all programs, meaning that while the graph represent the involvement in the study, the rectangular area bounded by a dot and origo does not represent the number of data points collected for all studies.

Comprehension of code can be measured in different ways. The most common is simply to ask control questions (for example like Tenny [94] or Börstler and Paech [13]), but there are more sophisticated methods employed, such as requiring the judges to find bugs (Fakhoury et al. [32]) or perform changes (Relf [77] and Lanna and Amyot [52]). These methods, commonly, serve to get an indication about to what extent the subject has accurately understood the code. 18 of the studies employ some kind of comprehension measurement: 16 of the 21 *Aspect* studies, and 2 of the 13 *Modelling* studies.

Effort of comprehension can be measured through proxies. The most common is to measure the time until a subject has indicated they have finished reading, or completed a task or answered a question. (ex. Scalabrino et al. [82] and Börstler and Paech [13]) More sophisticated methods include measuring effort by functional near-infrared brain imaging (Fakhoury et al. [34]).

A complete record of which studies employ what kind of measurements can be found in tables 4.2 and 4.1. These methodological aspects map directly to the components of the definitions.

Apart from experiments measuring *Impression*, *Comprehension* and *Effort*, studies use interviews to qualitatively capture aspects of readability. This method is used by Börstler et al. [14] and by Scalabrino et al. [83].

4.8.2 Correlating metrics

The correlation studies generally study the correlation between readability, on one hand, and some other metric on the other. To determine the strength of the relation, they usually apply statistical tests, but they differ in what tests they employ.

Mannan et al. employ the Kendall's τ and the Wilcoxon rank-sum test, but also calculate the effect size using Cohen's d . Wilcoxon rank-sum is also employed by Lee et al. [54, 53], while Kendall's τ is also employed by Scalabrino et al. [83].

Börstler et al. [12] and Coleman [21] both employ the Spearman rank coefficient ρ .

Grano et al. [38] use the Wilcoxon T-test to determine if the readability between automatically generated and manually written tests differ.

Fernandes et al. [36] and Alawad et al. [2] both use Pearson's r to determine the correlation for their respective relations. This may not be appropriate, because Pearson's r presume a linear relationship, and there is no clear evidence, or discussion on the reason to assume, that the relations examined are indeed linear.

Tashima et al. [92] plots the correlation, but do not employ any statistical tests to determine the strength of the relations.

4.8.3 Empirical methods summary

Answer to RQ2: The empirical studies examined in this master's thesis are of three kinds. The *Modelling* studies and the *Aspect* studies generally employ similar methods, in exposing subjects to code, and measuring the subjects' *impression*, *comprehension* and/or *effort* related to the code. The impression is usually measured on a Likert scale; comprehension most often with control questions, although a minority employ tasks or spotting similarities; and effort is most often measured using time as a proxy, although gaze tracking and functional near-infrared brain imaging is sometimes used as a complementary measure. Most of the *Modelling* studies do not measure *comprehension* or *effort*, while most of the *Aspect* studies do; compare table 4.1 and 4.2.

Correlation studies examine the relations between readability on one hand, and some other metric on the other. Generally, they employ some kind of statistical test to determine the correlation, including Kendall's τ , Wilcoxon rank-sum or Wilcoxon T or Spearman's ρ , but two studies use Pearson r , while one study does not employ any statistical test. See further sections 4.5 and 4.8.2 and table 4.4.

4.9 Developers' perspectives

To compare the state-of-the-art, as represented by software code readability research, with the state-of-practice in a software development community, interviews with software developers currently active in a major international consumer electronics company are conducted. In order to protect the privacy of the developers, they remain anonymous and have been given aliases in this thesis.

Commits explicitly mentioning readability are rare. In the Linux kernel, only about 0.8% commits match the search criterion, and in the repositories included in this study, only four could be found. For one of them, no contact could be made with either author or reviewers, and so, only three are included in this study. Also included are one senior and one junior developer. This selection is made “purposefully” (see sections 3.2.2 and 3.2.4).

The subjects all presented a willing and positive attitude towards the interviews. The second question (*What is readability to you?*, see section 3.2.2) successfully prompted reasoning about key elements of readability and common pain points in the experience of the respective developers. Most of the time, the developers finished after two or three, stating those were the factors they considered most closely connected to the concept of readability, and the answer to the question was concluded in a couple of minutes, even with additional prompts, but in one interview (labelled *Gamma* below), this question resulted in rich and far-reaching reasoning about the relevance of source code readability, and what constitutes precursors for readability and under what circumstances it can, in fact, be irrelevant.

In the cases with specific commits (the third question: *How does the change increase the readability?*, see section 3.2.2) the commit either reaffirms and exemplifies previously mentioned aspects of source code readability, or opens up new perspectives and reasoning.

The last two questions, about static analysis and readability metrics (see section 3.2.2), are questionnaire-like and prompt short and concise answers.

4.9.1 Case Delta

Delta is a developer with 13 years of experience from a large international consumer electronics company, and prior to that one year as a software development contractor, who has made a commit into a repository implementing a test harness.

An important aspect of readability, according to Delta, is the absence of large blocks of code. Large blocks should be divided by newlines to create logically coherent blocks, that are easier to read.

Another aspect important to this developer is naming. It is important, to Delta, that identifiers are accurately named after what they do or contain.

The change made by Delta is a commit to a Python 3 file in a repository implementing a test harness.

```

Improve camera error handling

camera_manager will now present a more
readable error when camera
is either missing or not calibrated.

```

Listing 4.1: *The commit message of Commit Delta. Used with permission.*

The commit message (listing 4.1) indicates that the code will produce a more readable error. Looking at the code change (listing 4.2), the developer explains that the code makes it more clear to the reader that this method depends on the previous initialization of a field (`self._camera`), and that the previous error message (the not-so-helpful `Attribute error: 'NoneType' object has no attribute 'capture_image'`), which did not describe the error, is replaced with a more explanatory `No camera configured`. This new error message should be more readable to humans and facilitate future configuration and maintenance of the system, says Delta.

```

async def _capture_image(self, step_id):
+   if not self._camera:
+       raise EnvironmentError('No camera configured.')
    return await self._camera.capture_image(
        self._output_dir, step_id)

```

Listing 4.2: *The diff of commit Delta. + denotes an added line. The original diff has been slightly shortened: in reality, a corresponding change was made in one more method. Used with permission.*

With regards to static analysis tools, Delta regularly use *lint*, *checkstyle* and *coverity*. This developer was not aware of any automated metrics specifically concerning readability.

4.9.2 Case Epsilon

Epsilon is a developer with 13 years of experience within a large international consumer electronics company. This developer is the author of a commit into a test framework API.

```

Add enum count as parameter to monitor functions

Change monitorAudio to take an enum const as
input to improve readability of TCs.

```

Listing 4.3: *The commit message of Commit Epsilon. The message has been slightly shortened: in reality, two methods were mentioned and affected by corresponding changes. Used with permission.*

Epsilon thinks it is important that code is kept simple, meaning that it avoids “too fanciful special language constructs”, and mentions the special comparison operators of Java Script as bad examples. It is also important that the code keeps a linear flow, and that, for example, callback structures should be avoided in favour of returns. Epsilon also mentions a dislike for event-based programming, for the same reason: the flow becomes nonlinear and hard to follow.

```

+  /**
+   * Enum for type of stream
+   *
+   * @readonly
+   */
+  const STREAM_TYPE = {
+    /** @member {string} LIVE_AUDIO
+     * The stream is an audio live stream */
+    LIVE_AUDIO: 'LIVE_AUDIO',
+
+    /** @member {string} AOD
+     * The stream is an Audio On Demand stream */
+    AOD: 'AOD',
+  }
+  Object.freeze(STREAM_TYPE)
+  ...
-  async monitorAudio(seconds, isLive = true) {
+  async monitorAudio(seconds, streamType) {
+    const isLive = streamType === STREAM_TYPE.LIVE_AUDIO
+    await this.sleep(2000) // Let audio playback start
+    if (!await this.isPlayingAudio()) {
+      await this.report(false, TEXTS.AUDIO_PLAY)
+    }
+    // Note: steps above takes 4 seconds ->
+    // adjust seconds a bit
+    await checkTimeAndRangeProgression(
+      this, Math.max(5, seconds - 2), isLive)
+  }

```

Listing 4.4: *The change in the API in commit Epsilon. The original diff has been slightly shortened: in reality two methods were affected by corresponding changes. The diff has also been slightly reformatted to fit the page width. + denotes an added line, while - denotes a removed line. ... denotes a block of redacted irrelevant code. Used with permission.*

This commit, according to its author, makes an implicit parameter explicit, and gives it a descriptive name. It also gives the parameter a descriptive type. (See listing 4.4) This, according to the author, provides more information to the reader, and makes the signature of the method clearer, enhancing readability of the client code. (See listing 4.5)

```

-  await testApi.monitorAudio(10)
+  await testApi.monitorAudio(10, STREAM_TYPE.LIVE_AUDIO)
+  ...
-  await testApi.monitorAudio(10, false)
+  await testApi.monitorAudio(10, STREAM_TYPE.AOD)

```

Listing 4.5: *An example of a diff in a test case using the changed API. + denotes an added line, while - denotes a removed line. ... denotes a redacted block of unchanged code. Used with permission.*

This developer regularly uses the static analysis tools *lint* and *coverity*, but does

not know of any automated readability metrics.

4.9.3 Case Lambda

Lambda was a reviewer of a commit into a large Android application. This developer has over 20 years of experience as a software developer, both as an employee at a major international telecommunications corporation, and as a software development contractor.

When it comes to readability, this developer thinks it is important not to create giant methods, but to make methods that makes it clear what's going in, and what's coming out. Naming is important. This developer thinks member variables should be avoided when possible, and especially ones declared static: it is important to keep the state space small. Documentation is important, but should be relevant and add value: avoid documenting what is already apparent from reading the code.

```
Cleanup navigation bar change

Just some minor cleanup to make the code
more readable and easier to understand.
```

Listing 4.6: *Commit message for commit Lambda. Used with permission.*

With regards to this particular commit, this developer, who served as a reviewer for the commit, thinks that the readability before and after this change is very similar. The naming change does not, in this reviewer's eyes, improve readability significantly, however, the positioning of parameters on separate lines holds an appeal. While the reviewer does not think it makes a big difference in this change, the moving of the `widthNotCoveredByScene` variable prompts reasoning about the importance of locality: variables should be declared as close to their use as possible, to reduce the need to scroll up and down a file.

```
-final int widthNotCoveredByScene = sDisplaySizeIncludingBars.x - sceneWidth;
  isTopDisplayAreaCoveredByScene = !equalsWithinTolerance(heightNotCoveredByScene,
    sStatusBarSize);
+final int widthNotCoveredByScene = sDisplaySizeIncludingBars.x - sceneWidth;
-final boolean isLeftOrRightDisplayAreaCoveredByScene = !equalsWithinTolerance(
-  widthNotCoveredByScene, sVisibleNavigationBarSeize);
+final boolean isCompleteHorizontalDisplayAreaCoveredByScene =
+  !equalsWithinTolerance(widthNotCoveredByScene, sVisibleNavigationBarSeize);
-updateOffsetBasedOnLeftOrRightLocatedNavBar(isLeftOrRightDisplayAreaCoveredByScene,
+updateOffsetBasedOnLeftOrRightLocatedNavBar(
+  isCompleteHorizontalDisplayAreaCoveredByScene,
  isTopDisplayAreaCoveredByScene, position);
```

Listing 4.7: *Diff for commit Lambda. Used with permission.*

This developer uses *checkstyle*, *findbugs* and *coverity* regularly, and sometimes employ *PMD*, but has not heard of any automated readability metrics.

4.9.4 Interview Gamma

Gamma is a developer and a software architect with 19 years of experience in the software industry at a major international consumer electronics company. Gamma is not interviewed about a specific commit, but for an insight into an architect's point of view.

In response to taking over the maintenance responsibility of several projects originating from teams outside of his organisation, Gamma has recently started to upgrade the importance of readability. This does not only relate to the concept of collaboration, but to an even bigger extent, it's for his own benefit. He finds himself looking at code thinking "Who wrote this?", realising it was himself. "If I think like that, how can I expect someone else to see what the code does?"

Gamma uses the concept *whirligig*¹ to describe a short piece of dense code, and says: "It doesn't matter if that part is unreadable. What matters is that it is encapsulated in a well defined, correctly named method, and that it can be tested, and ideally that it does not change state [of the program]." He elaborates that he sees unit tests as a part of readability, providing information about what the code is expected to do. If a *whirligig* needs to be changed, the test should catch any problems, and if it's properly contained, it doesn't need to be read or understood: just rewrite it. The readability resides in the function name or in a comment. A *whirligig* should not be in the middle of a function; it should be broken out and reside in a testable function of its own.

Naming is important, according to Gamma: code should be named after what it does, and do only what its name implies. Gamma spends a great deal of time pondering names of identifiers when writing code: to find names that work, linguistically. This also means following patterns and idioms, and to isolate functionality. As an example, he offers "If your class has an `open`-method, it has to have a `close` – even if it doesn't do anything."

Names should be distinct and precise. "What is a `Manager`? What is a `Handler`?", Gamma asks, rhetorically.

To reason about code, to describe it and define it becomes impossible, according to Gamma, when there are cyclic dependencies. This echoes of Thomas and Hunt [95, p. 192]. The same is true about code implementing too complex callback or event-based systems. Code should be linear and possible to follow. "`async/await` provides linearity of the code, and makes it possible to reason about the code (in a way callback structures does not), and the reasoning is more important than, or at least a precursor for, understanding."

Gamma regularly uses *checkstyle*, *findbugs* and *lint*, but does not know of any automated readability metric.

4.9.5 Interview Sigma

Sigma is a software developer with two years of experience in a large international consumer electronics company.

Sigma offers the opinion that readable code is short, concise, but still explanatory. Code should have a clear flow, though, and that is more important than having the least amount of lines of code possible.

For readability of a software system, a clear structure is important, meaning a well-thought structure of classes, their relations and the flow between them.

Descriptive and clear names of variables, classes and functions are important.

¹The interview was conducted in Swedish, and the word used was "snurra".

Sigma also talks about the expectations that are induced by the choice of language alone, and exemplifies by saying that a list comprehension in Python is to be expected, but that a similar piece of code in Java, using the `Stream-API` and lambda-expressions, is perceived harder to read. Sigma offers that the familiarity of the language may create a bias in this regard. Sigma is marginally more familiar with Python than with Java.

Documentation, according to Sigma, “is a crutch”. Complicated and intricate pieces of code can be explained with a comment, that increases readability.

Sigma regularly uses *lint*, but doesn't know of any automated readability metrics.

5.1 Madness to the method

Several of the included studies contain methodological problems that jeopard their results and conclusions. The three most common problems are flawed definitions, that lead to flawed measuring methods, an abundance of independent variables, risking chance correlations, and lack or rigour in subject selection.

5.1.1 Artefacts of a flawed definition

The definitions most commonly used by modelling studies (see section 4.6, tables 4.5 and 4.6) are along the lines of “an impression of the effort to understand”, yet the modelling studies tend to measure *impression*, but not *effort* nor *comprehension* (see table 4.1).

For a measurement of an impression to be productive, that impression needs to, in some way, predict the actual state of things; that is, in order for the measurement of “impression of the effort to understand” to be interesting, it has to be shown that this impression actually predicts or correlates to the actual “effort to understand”. To knowledge, two such efforts have been done. One by Scalabrino et al. [82], and this concludes that “readability [...] metrics are not correlated with understandability” [82, p. 425]. This is further indicated by Börstler and Paech in their correlation study: “Our results do not show any significant relationships between perceived readability and the other measures taken in the present study” concluding that “Perceived readability might therefore be insufficient as the sole measure of software readability or comprehension.” [13]

This is an important finding because it implies that the *impression* of the effort to understand does not correlate to the *actual* effort to understand, and if that is the case, this invalidates all studies into readability that measure only the *impression*, and not any proxy for *effort* or level of *understanding*. All but two of the 13 modelling studies suffer from this problem. (See table 4.1)

It could be argued that the impression of the effort to understand code is still a relevant measure, but the burden of proof lies with the model builders.

5.1.2 Riding shotgun

The studies in the empirical categories study the impact on the readability from a number of selected factors. The number of factors varies significantly: from twelve

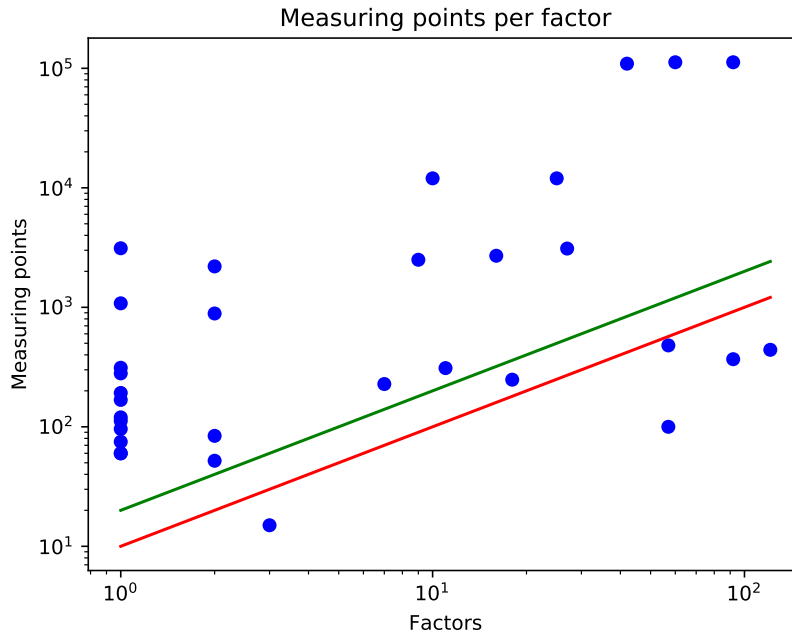


Figure 5.1: *Empirical studies with human subjects plotted with the number of factor they study against the number of human judgements performed. A red line marks 10 judgements per factor, and a green line marks 20 judgements per factor. (cf. Peduzzi et al. [74])*

papers selecting one or two factors, to five papers studying more than 90. Using a large number of independent variables can pose a problem since there is a risk that some will show a significant correlation by chance (see Kitchenham [49] and Courtney and Gustafson [23], who calls this phenomenon *Shotgun correlation*).¹

An approach to increase the statistical strength of a correlation is to increase the number of observations. Peduzzi et al. [74] proposes a rule of thumb of at least 10 to 20 as many samples as predictor variables. Figure 5.1 shows the empirical studies with regards to their sample sizes and the number of predictor variables, with the 10x and 20x lines plotted. Interestingly, just over 25% are below the 20x line, and almost 20% are below the 10x line. This means that there is a risk that their statistical strength is too weak to draw any conclusions.

This is common among the earliest studies, that were relatively small, and both DeYoung and Kampen [27] and Jørgensen [47] have relatively few measuring points per studied factor.

Two of the studies conducted by Scalabrino et al. [82, 83] study very many factors, and have only 5 and 4 measuring points per factor, respectively.

Newman et al. make a mini-survey with 5 subjects as a part of their theoretical reasoning on syntax restructuring, and this survey should be considered an indication,

¹A simple counting example: assume we accept a relation between two variables as valid at the customary $p < 0.05$. Over n dependant variables, we accumulate a theoretical upper bound of $p = 1 - 0.95^n$ to commit at least one type-1 error. For $n = 20$ this is $p = 1 - 0.95^{20} = 0.64$. To handle over 100 factors in the same model, tremendous statistical power in each relation is needed.

rather than a statistical foundation.

5.1.3 Subject selection

The kinds of subjects used for the studies can affect the contexts in which the results are valid. For the studies included in this thesis, 38 use human subjects to produce measuring points. Of those, 21 use students only, ten use a mix of students and professional developers, while only seven use professional developers exclusively. This bias is likely out of availability: most studies are conducted in the context of a university, and on campus, it is easier to find students than professional software developers. (See also figure 4.3 p. 28)

However, Wiese et al. [100] finds a difference between students and professional developers, in that they not only write code differently, but also perceive readability differently. This is further supported by Floyd et al. [37], who uses functional MRI to conclude that novices' and expert programmers' brains work differently when reading code.

This means that studies conducted on students may not be directly usable for the population of professional software developers, and vice versa, and that results of studies mixing students and professional software developers without proper care taken to separate the populations, may not apply to any of the populations.

5.2 Crumbling models

These three methodological problems could, at least partly, explain why the readability models fail to catch developer's effort to improve readability in code, as demonstrated by Pantiuchina et al. [71] and by Fakhoury et al. [33]. Lucas et al. [56] find that neither Buse and Weimer [9, 10], Posnett et al [75] nor Scalabrino et al. [84] capture the change in readability as introduced by lambdas, but also find that lambdas increase comprehension and human readability assessment.

Answer to RQ3: This study identifies three major risks to the validity of the studies examined:

- Some use a flawed definition allowing a methodological shortcut. This means they model *impression of* rather than *actual* effort of understanding, which is probably not the same thing. (See section 5.1.1).
- A fraction of the studies employ a *shotgun correlation* tactic, examining more dependent variables than their empirical data support. This risks finding chance relations. (See section 5.1.2)
- Although evidence suggests professionals and students perceive readability in fundamentally different ways, some studies mix professional and student subjects without discrimination. This risks the applicability of the studies. (See section 5.1.3)

All of these problems are most common in the *modelling* category of studies (see table 4.1 p. 17), but pose a problem also when those models are used in *correlation* studies (see table 4.4 p. 22).

Generally, the *aspect* studies (see table 4.2 p. 19) apply a higher methodological rigour, measuring *effort* to a much higher extent, sometimes using advanced methods, but their relevance remain limited, because they keep a narrow focus of factors, leading to a tendency to validate “common knowledge”, rather than to explore new factors.

5.3 Developers’ perspectives

To assess the completeness and relevance of the state-of-the-art, from an industrial practitioner’s perspective, research question 5 aims to compare developers’ experiences of software readability with the state-of-the-art. (See section 1.3) This is done by conducting qualitative interviews with practitioners (see sections 3.2.2 and 4.9). This section describes an analysis of the results of these interviews by comparing the factors mentioned by the developers during the interviews with the factors considered in the state-of-the-art, as represented by the papers included in this study.

All of the interviewed software professionals (see section 4.9) mention the importance of proper naming. Names should be distinct and descriptive. In this, they are in agreement with the state-of-the-art of research, where the study of *language* factors is common (see section 4.7 and table 4.8). Developers’ perception that naming is important is supported by studies like Marter et al. [58] and Fakhoury et al. [32, 34].

Sigma, Lambda and Gamma all talk about comments and documentation as a necessary evil, or a “crutch”. Comments should only be employed when the code is not self-explanatory, and such code is preferred. This perspective stands in contrast to the state-of-the-art, where comments are largely perceived to increase code readability. Aggarwal et al. go as far as to suggest that readability can be estimated by $CR = LOC/LOM$, where *LOC* is the number of lines of comments, and *LOM* is the total number of lines in the source code file [1, p. 235]. For these developers, comments do increase readability, but the presence of comments rather indicate a block of code with poor readability and the fewer comments needed, the better the readability of the surrounding code may be.

Delta talks about avoiding large blocks of code, breaking them up with empty lines. Line breaks are in this thesis categorized as *Layout* factors, but here it is also mentioned as a way to balance *Size* factors. Factors relating to *Size* are also lifted by Lambda (“giant methods”) and Sigma (“conciseness”). *Layout* is also mentioned by Lambda, who finds the separation of parameters to a function call on new lines increase readability.

Sigma, Epsilon and Gamma all mention the importance of a *linear flow* that is easy to follow. To some extent, this can seem to be connected to McCabe’s CC, and is then in the *Code* category of factors; but this might not capture the essence of the sentiment, because they contrast the linearity with callback patterns and event-based programming, that leads to complexities not captured by McCabes cyclomatic complexity measure. Such linearity, as contrasted by callback patterns, specifically,

is not directly studied in any of the included papers of this thesis.

Related may be the concept of structure, mentioned by Sigma. This is notably different from the *structural features* of Scalabrino et al. [84] and Dorn [28], and refers more to the design of the system and the execution flow akin to Börstler et al.'s use of the term [14]. This, in turn, is picked up by Gamma, who thinks that code with cyclic dependencies becomes hard to read and understand. These are factors belonging to the *Code* category, but represent a top-down perspective of reading that is not generally the focus of readability research in the papers included in this thesis.

Both Epsilon and Lambda mention the importance of clear signatures of functions, making it obvious what “goes in and comes out” of a function. This is related to Høst and Østvold's concept of “naming bugs”: bugs in, for example, function names, that contradict their signatures [43] and similar work of Liu et al [55]

Epsilon warns about “special language constructs” and Sigma talks about familiarity with the language at hand. These are important features affecting effort of comprehension, but are properties of the reader, rather than of the code, and may not be considered factors of readability as such.

Lambda talks about the importance of locality and scope of variables. These factors fall into the *Scope* category as studied by Chhabra [17] and Tashtoush et al. [93], but Lambda expands the concept in two directions. Lambda thinks that member variables should, if possible, be avoided, as they increase the state space of a class, making it harder to understand. This should be considered a factor in the *Code* category, and the impact of state spaces on readability has not been studied in any of the included papers. Lambda also says locality of variables is important as to avoid the need for scrolling. This is a physical factor having to do with *Size* factors, but also with *Presentation* – since different screens and editors can show different amounts of code at a time. Readability from the perspective *What fits on a screen?* has not been studied explicitly by any of the papers included in this study.

The importance of following idioms and patterns is lifted by Gamma. Code written in a familiar idiom or following a familiar pattern can be recognized by a reader, allowing a top-down approach to reading to a larger extent than functionally identical code not following a pattern. This is to some extent demonstrated by Börstler et al's Beauty and Beast [12], and Wiese et al. study “novice patterns” [100] but the identification of familiar and recommended patterns and idioms are not explicitly studied by any of the papers included in this study.

Gamma mentions tests, especially unit tests, as a part of readability. The tests provide information about what a function is expected to do, and as such they provide *Contextual* information, in the same fashion as comments and documentation do. This aspect of tests is not considered in any of the included papers.

Interesting to note is that none of the developers mentions indentation or white spaces. This is likely because all of the developers regularly use static analysis tools enforcing a coding style. When a coding style is uniformly enforced by a tool, the style problems become so rare that they are no longer considered at all. This means that while the research community still consider *Layout* factors, many of those are no longer an issue in practice.

While all of the developers regularly employ static analysis tools, none of them can name a readability metric. This indicates that automated readability indices have not (yet) made it into the tools employed by these developers. This, in turn, is

likely because such indices are still too immature to be of practical use.

Answer to RQ5: While many of the developers' experiences of readability are handled in the state-of-the-art, many others are not. The developers interviewed for this study seem to have a top-down approach to software source code reading, while much of the research into readability focuses on bottom-up reading.

The impact of programming language familiarity is mentioned by the developers. This is also mentioned in Collar and Valerdi [22], but is not studied in any of the included papers. The same holds for the effect of common idioms or patterns.

Concerning contextual information, developers seem to think that comments are a remedy for low readability, challenging the view from the research community that comments are indications of high readability, and at least one developer consider test cases contextual information – something that has been overlooked in the studies included.

5.4 Limitations and Threats to validity

This section provides the limitations and demarcations of this thesis, as well as the identified threats to its validity.

5.4.1 Database selection

Only one database, SCOPUS, is used for the initial search. Kitchenham points out “However, SCOPUS is particularly useful because it indexes publications from a large number of publishers, including ACM, IEEE, Elsevier, and Springer” [49]. This may mean that some relevant studies are not found. By employing backwards snowballing (cf. figure 3 of Kitchenham and Brereton [50]), the inclusion of relevant papers outside of the SCOPUS database is ensured.

5.4.2 Search parameters

The search parameters, both in SCOPUS and in the software repositories, are limited to papers and commits explicitly mentioning *readability*. Bordering concepts, such as *understandability*, *comprehensibility* and *legibility* are intentionally left out. This is done for two reasons: because demarcation becomes increasingly difficult when more concepts are included (why include *comprehensibility*, but not *complexity*?); and because this master's thesis explicitly aims to map *readability* of software source code.

The search was based on abstract and titles only. This has probably not limited the inclusion of papers, rather it has become indicated that it is popular to mention *readability* in abstracts of papers not otherwise concerned with the topic.

5.4.3 Software repositories

The software repositories searched are only those in which the sponsoring company has active developers. This means that most repositories are proprietary and not

open for public scrutiny. Furthermore, it means that internal company culture and policies may bias the language used to describe certain changes, and this might affect the use of language in the commit messages.

5.4.4 Qualitative methods, subject selection and transferability

Subjects eligible for interviews are, or have been, employees or contractors of the sponsoring company. This means that internal company culture may affect the way they regard readability and neighbouring concepts, and the way they think of code. This bias may, or may not, separate them from a wider software development community.

This means that the results of the interviews may not be transferable or provide a general representation of practitioners' perspectives and experience of software source code readability, and neither is this the purpose of the interviews. The purpose, rather, is to provide a sounding board and an indication of the completeness and generalizability of the state-of-the-art.

5.4.5 Single researcher

The literature study was conducted by a single researcher, meaning that some papers may have been included or excluded incorrectly, or that some papers may have been assigned an incorrect classification. (cf. Kitchenham [49]). Literature studies' rigour increase with the number of researchers, and parts of the processes described by Kitchenham and Brereton [50] have not been observed, because they require discussion within a team of researchers.

Chapter 6

Conclusions and Future Work

This study finds that many studies endeavouring to model software source code readability suffer from theoretical and methodological problems that make them theoretically unsound, and this is confirmed in empirical studies. This makes not only the models as such unreliable, but also jeopardises the validity of the results of correlation studies based upon output from any of the unsound models.

When defining the concept of readability in the context of software source code, the included papers employ three central concepts: *impression*, *effort* and *comprehension*. 14 of the papers explicitly state definitions, and all but one of these use at least one of these concepts. Seven mention *effort of understanding*, and five of those also prefix *impression*. (See section 4.6) Measuring only impression seems to give a flawed image of actual readability. (See section 5.1.1)

Factors considering impacting readability can be ordered into the eight categories *Language*, *Layout*, *Presentation*, *Size*, *Code*, *Scope*, *Context* and *Entropy*. These categories capture many aspects of readability of software source code.

While the studies included in this systematic literature review have examined aspects of readability of source code, none perform a qualitative study of developers' experiences or perceptions of readability. Even a minimal qualitative interview study, such as the one performed in this master's thesis, uncover additional factors and aspects not considered in any of the included studies.

For future research to become more relevant to the software development industry and software development practitioners, further research into how developers read code is needed. In order to do this, qualitative methods should be employed to ensure an early inclusion of developers' perspectives and strategies to read and understand software source code in different settings and roles.

It is apparent that the terminology of the field is underdeveloped. The clearest example is the term "structural" features (see sections 4.7 and 5.3), that is given at least two disjunct meanings. A systematic inventory of terms used in research on software source code readability and nearby fields, such as for example program comprehension, would lower the risk of confusion and lay a foundation for further advancement of the fields.

List of Tables

4.1	Modelling studies	17
4.2	Aspect studies	19
4.3	Theoretical studies	20
4.4	Correlation studies	22
4.5	Defintions, their origins and their use	23
4.6	Concepts of readability and their inclusion in defintions	24
4.7	Categories of factors affecting readability	24
4.8	Factor category use in studies	26

List of Figures

1.1	Elements of texts and their relations	4
4.1	Hierarcical categorization of studies	15
4.2	Publishing timeline of included studies	16
4.3	Plot of sizes of included empirical studies with regards to programs analysed, subjects and kinds of subjects	28
5.1	Plot of subjects against factors considered, with lines marking 10x and 20x	38

References

- [1] K.K. Aggarwal, Y. Singh, and J.K. Chhabra. An integrated measure of software maintainability. In *Proceedings of the Annual Reliability and Maintainability Symposium*, pages 235–241, 2002.
- [2] D. Alawad, M. Panta, M. Zibran, and M.R. Islam. An empirical study of the relationships between code readability and software complexity. In *27th International Conference on Software Engineering and Data Engineering, SEDE 2018*, pages 122–127, 2018.
- [3] M. Aragão, M.J. Varanda Pereira, and P.R. Henriques. Scaling up a programmers’ profile tool. In *OpenAccess Series in Informatics*, volume 74, 2019.
- [4] Ronald Baecker. Enhancing program readability and comprehensibility with tools for program visualization. In *Proceedings - International Conference on Software Engineering*, pages 356–366, 1988.
- [5] A. Batool, M.H. Rehman, A. Khan, and A. Azeem. Impact and comparison of programming constructs on Java and C# source code readability. *International Journal of Software Engineering and its Applications*, 9(11):79–90, 2015.
- [6] D. Beyer and A. Fararoyy. A simple and effective measure for complex low-level dependencies. In *IEEE International Conference on Program Comprehension*, pages 80–83, 2010.
- [7] D. Binkley, M. Davis, D. Lawrie, and C. Morrell. To camelcase or under-score. In *IEEE International Conference on Program Comprehension*, pages 158–167, 2009.
- [8] Oliver Burn. checkstyle, 2017. <http://checkstyle.sourceforge.net>.
- [9] R.P.L. Buse and W.R. Weimer. A metric for software readability. In *ISSTA’08: Proceedings of the 2008 International Symposium on Software Testing and Analysis 2008*, pages 121–130, 2008.
- [10] R.P.L. Buse and W.R. Weimer. Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4):546–558, 2010.
- [11] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. The effect of identifier naming on source code readability and quality. In *ESEC/FSE Doctoral Symposium’09 - Proceedings of the Doctoral Symposium for ESEC/FSE*, pages 33–34, 2009.
- [12] J. Börstler, M.E. Caspersen, and M. Nordström. Beauty and the beast: on the readability of object-oriented example programs. *Software Quality Journal*, 24(2):231–246, 2016.

- [13] J. Börstler and B. Paech. The role of method chains and comments in software readability and comprehension—an experiment. *IEEE Transactions on Software Engineering*, 42(9):886–898, Sep. 2016.
- [14] J. Börstler, H. Störrle, D. Toll, J. Van Assema, R. Duran, S. Hooshangi, J. Jeuring, H. Keuning, C. Kleiner, and B. MacKellar. “I know it when I see it” – Perceptions of code quality: ITiCSE’17 working group report. In *ITiCSE-WGR 2017 - Proceedings of the 2017 ITiCSE Conference on Working Group Reports*, volume 2018-January, pages 70–85, 2018.
- [15] Patrick Cadwell. Readability and controlled language. Master’s thesis, Dublin City University, 2008.
- [16] C. Chaweewan, A. Surarerks, A. Rungsawang, and B. Manaskasemsak. Development of programming capability framework based on aptitude and skill. In *2018 3rd International Conference on Computer and Communication Systems, ICCCS 2018*, pages 444–447, 2018.
- [17] J.K. Chhabra. Code cognitive complexity: A new measure. In *Proceedings of the World Congress on Engineering 2011, WCE 2011*, volume 2, pages 1249–1253, 2011.
- [18] S. Choi, S. Kim, J. Kim, and S. Park. Metric and tool support for instant feedback of source code readability. *Tehnicki Vjesnik*, 27(1):221–228, 2020.
- [19] S. Choi, S. Kim, J.-H. Lee, J.A. Kim, and J.-Y. Choi. Measuring the extent of source code readability using regression analysis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 10963 LNCS:410–421, 2018.
- [20] Meri Coleman and Ta Lin Liau. A computer readability formula designed for machine scoring. *Journal of Applied Psychology*, 60(2):283, 1975.
- [21] R. Coleman. Aesthetics versus readability of source code. *International Journal of Advanced Computer Science and Applications*, 9(9):12–18, 2018.
- [22] Emilio Collar Jr and Ricardo Valerdi. Role of software readability on software development cost. In *21st Forum on COCOMO and Software Cost Modeling*, October 2006.
- [23] Richard E Courtney and David A Gustafson. Shotgun correlations in software measures. *Software Engineering Journal*, 8(1):5–13, 1993.
- [24] D. Crookes. Generating readable software. *Software engineering journal*, 2(3):64–70, 1987.
- [25] F. Deißeböck and M. Pizka. Concise and consistent naming. In *Proceedings - IEEE Workshop on Program Comprehension*, pages 97–106, 2005.
- [26] F. Deißeböck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.
- [27] G.E. DeYoung and G.R. Kampen. Program factors as predictors of program readability. In *Proceedings - International Computer Software and Applications Conference*, pages 668–673, 1979.

- [28] Jonathan Dorn. A general software readability model. Master's thesis, University of Virginia, Charlottesville, 2012. <http://www.cs.virginia.edu/~weimer/students/dorn-mcs-paper.pdf>).
- [29] R.M. Dos Santos and M.A. Gerosa. Impacts of coding practices on readability. In *Proceedings - International Conference on Software Engineering*, pages 277–285, 2018.
- [30] William H DuBay. The principles of readability. *Online Submission*, 2004.
- [31] James L Elshoff and Michael Marcotty. Improving computer program readability to aid modification. *Communications of the ACM*, 25(8):512–521, 1982.
- [32] S. Fakhoury, Y. Ma, V. Arnaoudova, and O. Adesope. The effect of poor source code lexicon and readability on developers' cognitive load. In *Proceedings - International Conference on Software Engineering*, pages 286–296, 2018.
- [33] S. Fakhoury, D. Roy, A. Hassan, and V. Arnaoudova. Improving source code readability: Theory and practice. In *IEEE International Conference on Program Comprehension*, volume 2019-May, pages 2–12, 2019.
- [34] S. Fakhoury, D. Roy, Y. Ma, V. Arnaoudova, and O. Adesope. Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization. *Empirical Software Engineering*, 2019.
- [35] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachsel, M. Papendieck, T. Leich, and G. Saake. Do background colors improve program comprehension in the #ifdef hell? *Empirical Software Engineering*, 18(4):699–745, 2013.
- [36] E. Fernandes, L.P. Ferreira, E. Figueiredo, and M.T. Valente. How clear is your code? an empirical study with programming challenges. In *CIBSE 2017 - XX Ibero-American Conference on Software Engineering*, pages 43–56, 2017.
- [37] Benjamin Floyd, Tyler Santander, and Westley Weimer. Decoding the representation of code in the brain: An fmri study of code review and expertise. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 175–186. IEEE, 2017.
- [38] G. Grano, S. Scalabrino, H.C. Gall, and R. Oliveto. An empirical investigation on the readability of manual and generated test cases. In *Proceedings - International Conference on Software Engineering*, pages 348–351, 2018.
- [39] Robert Gunning. *Technique of clear writing*. McGraw-Hill, 1952.
- [40] Maurice H. Halstead. *Elements of Software Science*. Elsevier North-Holland, Amsterdam, 1977.
- [41] G.J. Holzmann. Code clarity. *IEEE Software*, 33(2):22–25, 2016.
- [42] H. Hourani, H. Wasmi, and T. Alrawashdeh. A code complexity model of object oriented programming (oop). In *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology, JEEIT 2019 - Proceedings*, pages 560–564, 2019.
- [43] E.W. Høst and B.M. Østvold. Debugging method names. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 5653 LNCS:294–317, 2009.

- [44] A. Jaaksi. Implementing interactive applications in C++. *Software: Practice and Experience*, 25(3):271–289, 1995.
- [45] L. Jiang, H. Liu, and H. Jiang. Machine learning based recommendation of method names: How far are we. In *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, pages 602–614, 2019.
- [46] J. Johnson, S. Lubo, N. Yedla, J. Aponte, and B. Sharif. An empirical study assessing source code readability in comprehension. In *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, pages 513–523, 2019.
- [47] Anker Helms Jørgensen. A methodology for measuring the readability and modifiability of computer programs. *BIT Numerical Mathematics*, 20(4):393–405, 1980.
- [48] J Peter Kincaid, Robert P Fishburne Jr, Richard L Rogers, and Brad S Chissom. Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel. Technical report, 1975.
- [49] B. Kitchenham. What’s up with software metrics? - a preliminary mapping study. *Journal of Systems and Software*, 83(1):37–51, 2010.
- [50] B. Kitchenham and P. Brereton. A systematic review of systematic review process research in software engineering. *Information and Software Technology*, 55(12):2049–2075, 2013.
- [51] C. Kussmaul. Novel language syntax to enhance readability : White space, parameter sets, & control structures. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pages 767–768, 2008.
- [52] M. Lanna and D. Amyot. Spotting the difference. *Software - Practice and Experience*, 41(6):607–626, 2011.
- [53] T. Lee, J.-B. Lee, and H.P. In. Effect analysis of coding convention violations on readability of post-delivered code. *IEICE Transactions on Information and Systems*, E98D(7):1286–1296, 2015.
- [54] T. Lee, J.B. Lee, and H.P. In. A study of different coding styles affecting code readability. *International Journal of Software Engineering and its Applications*, 7(5):413–422, 2013.
- [55] K. Liu, D. Kim, T.F. Bissyande, T. Kim, K. Kim, A. Koyuncu, S. Kim, and Y. Le Traon. Learning to spot and refactor inconsistent method names. In *Proceedings - International Conference on Software Engineering*, volume 2019-May, pages 1–12, 2019.
- [56] W. Lucas, R. Bonifácio, E.D. Canedo, D. Marcílio, and F. Lima. Does the introduction of lambda expressions improve the comprehension of java programs? In *ACM International Conference Proceeding Series*, pages 187–196, 2019.
- [57] U.A. Mannan, I. Ahmed, and A. Sarma. Towards understanding code readability and its impact on design quality. In *NL4SE 2018 - Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering, Co-located with FSE 2018*, pages 18–21, 2018.

- [58] T. Marter, P. Babucke, P. Lembken, and S. Hanenberg. Lightweight programming experiments without programmers and programs: An example study on the effect of similarity and number of object identifiers on the readability of source code using natural texts. In *Onward! 2016 - Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 1–14, 2016.
- [59] Robert C. Martin. *Clean Code*. Prentice Hall, 2009.
- [60] Jennifer Mason. *Qualitative Researching*. SAGE Publications Inc., third edition, 2017.
- [61] Joseph A. Maxwell. Understanding and Validity in Qualitative Research. In A. Michael Huberman and Matthew B. Miles, editors, *The Qualitative Researcher's Companion*, pages 36–64. SAGE Publications Inc., Thousand Oaks, 2011.
- [62] Thomas J McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2:308–320, December 1976.
- [63] G Harry McLaughlin. Smog grading-a new readability formula. *Journal of reading*, 12(8):639–646, 1969.
- [64] Q. Mi, J. Keung, X. Mei, Y. Xiao, and W.K. Chan. A gamification technique for motivating students to learn code readability in software engineering. In *Proceedings - 2018 International Symposium on Educational Technology, ISET 2018*, pages 250–254, 2018.
- [65] Q. Mi, J. Keung, Y. Xiao, S. Mensah, and Y. Gao. Improving code readability classification using convolutional neural networks. *Information and Software Technology*, 104:60–71, 2018.
- [66] John Moffitt and Robert J. Sandler. The Year 2000 FAQ, 1998. <https://www-users.cs.umn.edu/shekhar/5180/y2kfaq.txt>.
- [67] N/A. pmd, 2020. <https://pmd.github.io/>.
- [68] C.D. Newman, B. Bartman, M.L. Collard, and J.I. Maletic. Simplifying the construction of source code transformations via automatic syntactic restructurings. *Journal of Software: Evolution and Process*, 29(4), 2017.
- [69] Ankit Pahal and Rajender Chillar. Code readability: A review of metrics for software quality. *International Journal of Computer Trends and Technology*, 46:1–4, 04 2017.
- [70] J.D. Palmer and E. Hillenbrand. Reimagining literate programming. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA*, pages 1007–1014, 2009.
- [71] J. Pantuichina, M. Lanza, and G. Bavota. Improving code: The (mis) perception of quality metrics. In *Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018*, pages 80–91, 2018.
- [72] A. Parrish, D. Cordes, R. Borie, and S. Edara. Illustrating client and implementation readability tradeoffs in Ada and C++. *Software - Practice and Experience*, 26(7):799–814, 1996.

- [73] L. Pascarella, M. Bruntink, and A. Bacchelli. Classifying code comments in java software systems. *Empirical Software Engineering*, 24(3):1499–1537, 2019.
- [74] Peter Peduzzi, John Concato, Elizabeth Kemper, Theodore R Holford, and Alvan R Feinstein. A simulation study of the number of events per variable in logistic regression analysis. *Journal of clinical epidemiology*, 49(12):1373–1379, 1996.
- [75] D. Posnett, A. Hindle, and P. Devanbu. A simpler model of software readability. In *Proceedings - International Conference on Software Engineering*, pages 73–82, 2011.
- [76] Bill Pugh and David Hovemeyer. findbugs, 2005. <http://findbugs.sourceforge.net/>.
- [77] P.A. Relf. Tool assisted identifier naming for improved software readability: An empirical study. In *2005 International Symposium on Empirical Software Engineering, ISESE 2005*, pages 53–62, 2005.
- [78] Talita Vieira Ribeiro. Alinhando perspectivas de qualidade em código fonte a partir de estudos experimentais-um caso na indústria. Master thesis, MS thesis, PESC/-COPPE, Universidade Federal do Rio de Janeiro, Rio de . . . , 2014.
- [79] T.V. Ribeiro and G.H. Travassos. Who is right? evaluating empirical contradictions in the readability & comprehensibility of source code. In *CIBSE 2017 - XX Ibero-American Conference on Software Engineering*, pages 99–112, 2017.
- [80] Jr. Rodrigues, E. and L. Montecchi. Towards a structured specification of coding conventions. In *Proceedings of IEEE Pacific Rim International Symposium on Dependable Computing, PRDC*, volume 2019-December, pages 168–177, 2019.
- [81] Johannes Sametinger. A tool for the maintenance of C++ programs. In *Conference on Software Maintenance*, pages 54–59, 1990.
- [82] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto. Automatically assessing code understandability: How far are we? In *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 417–427, 2017.
- [83] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto. Automatically assessing code understandability. *IEEE Transactions on Software Engineering*, 2019.
- [84] S. Scalabrino, M. Linares-Vasquez, D. Poshyvanyk, and R. Oliveto. Improving code readability models with textual features. In *IEEE International Conference on Program Comprehension*, volume 2016-July, 2016.
- [85] S. Scalabrino, M. Linares-Vásquez, R. Oliveto, and D. Poshyvanyk. A comprehensive model for code readability. *Journal of Software: Evolution and Process*, 30(6), 2018.
- [86] T. Sedano. Code readability testing, an empirical study. In *Proceedings - 2016 IEEE 29th Conference on Software Engineering Education and Training, CSEEdT 2016*, pages 111–117, 2016.
- [87] RJ Senter and Edgar A Smith. Automated readability index. Technical report, CINCINNATI UNIV OH, 1967.

- [88] B. Sharif and J.I. Maletic. An eye tracking study on camelcase and under-score identifier styles. In *IEEE International Conference on Program Comprehension*, pages 196–205, 2010.
- [89] T. Siddiqui and A. Ahmad. Mining software repositories for software metrics (msr-sm): Conceptual framework. *International Journal of Innovative Technology and Exploring Engineering*, 8(10):4173–4177, 2019.
- [90] D. Silva, N. Tsantalis, and M.T. Valente. Why we refactor? confessions of github contributors. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 13-18-November-2016, pages 858–870, 2016.
- [91] Jonathan M Smith. The software design laboratory. *Computing Systems*, 4(4):385–404, 1991.
- [92] K. Tashima, H. Aman, S. Amasaki, T. Yokogawa, and M. Kawahara. Fault-prone java method analysis focusing on pair of local variables with confusing names. In *Proceedings - 44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018*, pages 154–158, 2018.
- [93] Y. Tashtoush, Z. Odat, I. Alsmadi, and M. Yatim. Impact of programming features on code readability. *International Journal of Software Engineering and its Applications*, 7(6):441–458, 2013.
- [94] T. Tenny. Program readability: procedures versus comments. *IEEE Transactions on Software Engineering*, 14(9):1271–1279, Sep. 1988.
- [95] Dave Thomas and Andy Hunt. *The Pragmatic Programmer*. Addison-Wesley, 20th anniversary edition, 2020.
- [96] United Nations. About the sustainable development goals, 2020. <https://www.un.org/sustainabledevelopment/sustainable-development-goals/>.
- [97] Muhammad Usman, Ricardo Britto, Jürgen Börstler, and Emilia Mendes. Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. *Information and Software Technology*, 85:43 – 59, 2017.
- [98] June M Verner, O Pearl Breerton, Barbara A Kitchenham, Mark Turner, and Mahmood Niazi. Systematic literature reviews in global software development: A tertiary study. In *Proceedings of the EASE 2012*. IET, 2012.
- [99] Kurt D Welker, Paul W Oman, and Gerald G Atkinson. Development and application of an automated source code maintainability index. *Journal of Software Maintenance: Research and Practice*, 9(3):127–159, 1997.
- [100] E.S. Wiese, A.N. Rafferty, and A. Fox. Linking code readability, structure, and comprehension among novices: It’s complicated. In *Proceedings - 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training, ICSE-SEET 2019*, pages 84–94, 2019.
- [101] A. Wulff-Jensen, K. Ruder, E. Triantafyllou, and L.E. Bruni. Gaze strategies can reveal the impact of source code features on the cognitive load of novice programmers. *Advances in Intelligent Systems and Computing*, 775:91–100, 2019.

