Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

# A Validation of Cognitive Complexity as a Measure of Source Code Understandability

Marvin Muñoz Barón

| | |
|---|---|
| **Course of Study:** | Softwaretechnik |
| **Examiner:** | Prof. Dr. Stefan Wagner |
| **Supervisor:** | Marvin Wyrich, M.Sc. |
| **Commenced:** | April 30, 2019 |
| **Completed:** | October 30, 2019 |

## Abstract

Understanding source code is an integral part of the software development process. There have been numerous attempts to describe source code metrics that correlate with understandability, but few are empirically evaluated and those that are, show no meaningful support for their purpose. A recent effort is cognitive complexity, a measure specifically described as a metric for understandability. The primary goal of this study was to find evidence towards determining the validity of cognitive complexity as a measure of source code understandability. To achieve this, we planned and executed a systematic literature search to find source code snippets that were evaluated in terms of understandability in previous studies. The cognitive complexity of these snippets was then correlated with the measures of understandability used in the studies. The literature search identified data from 14 studies spanning over 324 code snippets and approximately 24,400 individual human evaluations, which were used in the correlation analysis. The results show that for most of the measures from existing studies, cognitive complexity significantly correlates with source code understandability. The mean of the significant correlations was 0.654 for comprehension time and 0.411 for subjective ratings of understandability, readability, and confidence by the study participants. The correlation with the correctness of the comprehension tasks showed mixed results, with some significant positive and some significant negative correlations. Overall, the evidence gathered in this study shows significant support for the validity of cognitive complexity. As far as we know, cognitive complexity is the first solely code-based metric that correlates with source code understandability in a meaningful way.

# Contents

# List of Figures

# List of Tables

# List of Code Snippets

# Acronyms

**ACM** Association for Computer Machinery. 26

**DID** data set identifier. 37

**EDA** electrodermal. 18

**EEG** electroencephalogram. 18

**EiPE** explain in plain English. 18

**fMRI** functional magnetic resonance imaging. 17

**IEEE** Institute of Electrical and Electronics Engineers. 26

**LOC** lines of code. 19

**MCC** McCabe's Cyclomatic Complexity. 19

**PL** programming language. 37

**PNo** number of participants. 37

**SNo** number of code snippets. 37

# 1 Introduction

Ever since the first computer programs were written, understanding what the purpose of a snippet of source code is has been an integral part of software development. Thirty years ago, program understanding was called the challenge for the 1990s [Cor89]. But even nowadays professional developers spend more than 55% of their time on activities related to program comprehension [MML15; XBL+18]. Through static code analysis and code-oriented software measures, developers always seek to optimize ways to read and work with source code. Numerous metrics exist to assess software quality but none seem to significantly correlate with code understandability [FALK11; SBV+19].

A recent effort to find this missing link between software measures and understandability is the newly introduced **cognitive complexity** measure, described by Campbell et al. as a metric for understandability [Cam18]. The goal of this work is to evaluate whether cognitive complexity correlates with measures for understandability from existing studies. Through a systematic literature search, code snippets of which the understandability was previously measured, were identified. Cognitive complexity was then calculated for these snippets and their values were correlated with the measured variables for understandability.

To summarize, the main contributions of this work are:

- The planning, execution, and results of a systematic literature search for studies that measure program comprehension from a human developer's point of view.
- A comprehensive analysis of cognitive complexity as a measure of source code understandability. To this end, 14 studies spanning over 324 code snippets and approximately 24,400 individual human evaluations were used to investigate the correlation of cognitive complexity with measures for understandability.

**Structure**: This report is organized as follows. Chapter 1 introduces the core concepts of source code understandability, gives an overview of related works in the field, and describes the core metric of this work, cognitive complexity. In Chapter 2, the methodology of the literature search and correlation analysis is presented. Chapter 3 shows the results of the correlation analysis. A detailed discussion of these results and an explanation of any threats to the validity to this work follows in Chapter 4. Finally, Chapter 5 summarizes the contents of the thesis, provides a conclusion and directions for future research.

## 1.1 Source Code Understandability

There have been numerous attempts to describe what code understandability is and what other software quality attributes could be considered as its influencing factors. Boehm et al. [BBL76] describe understandability as an attribute of maintainability composed of consistency, self-descriptiveness,

structuredness, conciseness, and legibility. Specifically, they define that "code possesses the characteristic of understandability to the extent that its purpose is clear to the inspector". An excerpt from their quality model can be seen in Figure 1.1 which shows understandability and its related attributes.

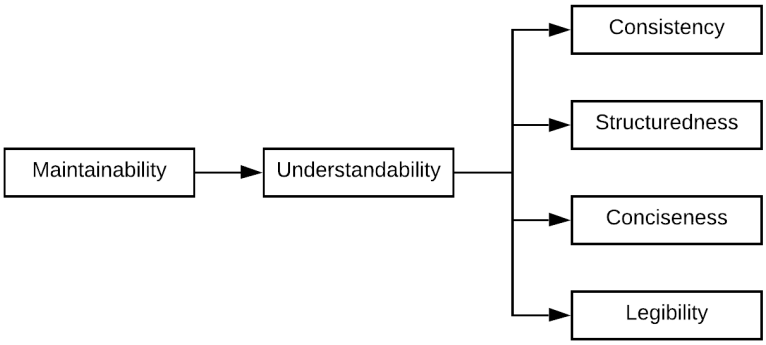


**Figure 1.1:** Understandability and related attributes as described by Boehm et al. [BBL76]

In the ISO 9126 standard [IEC01], understandability has moved to be an attribute of usability and is described as the "capability of the software product to enable the user to understand whether the software is suitable, and how it can be used for particular tasks and conditions of use". The type of understandability described here clearly is not code understandability as described by Boehm et al. but instead, the understandability of the software from the end user's perspective.

With the ISO 25010 [Iso11] standard, understandability has completely vanished as a quality attribute. The attribute that most closely resembles traditional notions of code understandability would be analyzability, which is defined as the "degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified". While the definition for analyzability captures the broader impacts of understandability, we find that it fails to describe the essence of understanding, which is deriving meaning from text or source code. Therefore we deem that it is not a perfect substitute definition for understandability.

### 1.1.1 Definitions

There is a need for clear definitions of the terms surrounding understandability. For this work, we use the original definition of understandability by Boehm et al. [BBL76]:

**Definition 1.1.1 (Understandability)**
*Code possesses the characteristic of understandability to the extent that its purpose is clear to the inspector.*

Additionally, the terms understandability and comprehensibility, as well as understanding and comprehension, will be used synonymously. Another difficulty regarding terminology is the close relation of readability and understandability. Boehm et al. state that legibility is necessary for understandability [BBL76]. Some studies explicitly separate understandability from readability. Scalabrino et al. stress that while a developer might consider a piece of code readable, they might

still have difficulties understanding it [SBV+19]. Unfortunately, there is not a field-wide standard and in some other works, the distinction is not as clear-cut. For example, code readability is described as the amount of mental effort required to understand the code by Sedano [Sed16]. In their readability study, Buse et al. advised their participants that, while not formally defining it, "readability is [their] judgment about how easy a block of code is to understand" [BW10].

In this study, readability is defined as follows:

**Definition 1.1.2 (Readability)**
*Readability is the human judgment of the perceived reading difficulty of a given snippet of code.*

Furthermore, for this work, complexity is separated from readability and understandability. We use the definition of software complexity by Ramamoorthy et al. [RTYB85]:

**Definition 1.1.3 (Complexity)**
*Software complexity is the degree or difficulty in analysis, testing, design and implementation of software.*

### 1.1.2 Measuring Understandability

When designing an experiment to measure understandability, it is important to design the tasks to be performed by the subjects in a way that mirrors the understanding of code by professionals in the real world. Siegmund [Sie16] describes three distinct ways of measuring understandability in studies. First, they mention think-aloud protocols, where the subject's thoughts are recorded, transcribed and later analyzed to gain knowledge about their thought process during the comprehension. Another way of capturing comprehension they described is through memorization activities, where developers are asked to recall source code that they have previously seen. Another possibility with which studies have measured comprehension is through so-called comprehension tasks. An example mentioned was an activity where the subjects had to fill out blank lines to complete source code. Another widely used way of testing understanding is a series of comprehension questions, where participants are asked to calculate the output of a program by hand or describe the program's functionality in words.

One also has to consider how to operationalize understandability into appropriate proxy variables. Usually, when conducting the experiment, different types of measures are taken to assess the degree with which a program was understood. Most record whether the comprehension task was completed successfully [KW13; WDS81; WR99]. Others also record the time taken to answer questions or perform a task on the code such as locating or fixing a bug [FALK11; HSH17]. This is not the norm however since some studies preset the amount of time which the subjects have to complete the comprehension task [AMS12; RW97; WRSC99].

A newer trend in studies examining code understandability is the use of physiological measures. In 2014, Siegmund et al. [SKA+14] conducted the first code comprehension study inside a functional magnetic resonance imaging (fMRI) scanner. They investigated activation patterns in different brain regions while participants located syntax errors in and calculated the output of small code snippets. Their work represents a first exploratory effort in lowering the barriers for fMRI studies in software engineering and they reported consistent results in a replication study. Floyd et al. [FSW17] compared the brain activity during code review, comprehension, and prose review tasks inside an

fMRI scanner. Their work was directly influenced by the work of Siegmund et al. They found that the neural representations of programming and natural languages are distinctly different but greater skill and expertise make this difference less pronounced.

Fucci et al. [FGN+19] conducted a replication study of the work by Floyd et al. using biometrics sensors such as electroencephalogram (EEG), electrodermal (EDA) and heart-related sensors. They then used machine learning to automatically identify the type of task performed by the subjects. They concluded that lightweight biometric sensors can recognize the type of comprehension task. Yeh et al. [YGYZ17] used an EEG to compare the brain activity for the comprehension of short C and C++ code snippets. The outcome of their study was that the average magnitude in broad alpha and theta bands is positively correlated with the cognitive workload when comprehending code snippets. They suggest that these results could be used in intelligent tutoring systems.

Turner et al. [TFSL14] examined the comprehension of C++ and Python code with an eye-tracking device. They compared the accuracy, speed, and visual effort while describing the functionality of and finding bugs in code snippets. The obtained results showed that while there was no statistical difference for speed and accuracy, they found significant distinctions in the fixation rate of buggy lines between the two languages. Fritz et al. [FBM+14] used EDA, EEG and an eye-tracking device to predict the perceived difficulty of comprehension tasks. They found that using their data, a Naive Bayes classifier could be trained to detect whether a task would be found difficult with a precision higher than 70% and a recall over 62%.

### 1.1.3 Related Work

While many papers describe theoretical models for program comprehension and metrics that support these models, in practice, few are evaluated in empirical experiments. There have been some efforts to correlate software metrics with measures of understandability.

Kasto et al. [KW13] attempted to find a connection between a multitude of different code metrics and comprehension difficulty by analyzing the student answers of a final exam in an introductory Java programming course. The comprehension tasks included code tracing tasks, where students figured out the output of the source code by hand and explain in plain English (EiPE) questions, where they explained the functionality of the code in words. They found that some of the investigated metrics, cyclomatic complexity, nested block depth, and two dynamic metrics significantly correlated with the student performance in code tracing exam questions. No significant correlation could be found with EiPE questions.

Peitek et al. [PSA+18] investigated the correlation of software metrics with the brain deactivation strength, used as an indicator of concentration levels, captured while performing comprehension tasks in an fMRI scanner. They found that for all deactivated areas, higher values for DepDegree and Halstead's measures indicated more concentration. For cyclomatic complexity, they found that higher values seemed to show less concentration for participants. Lines of code showed the weakest correlations. They did warn, however, that the small sample size of the study and due to the snippets not being designed to answer this question, the results might not necessarily be statistically significant despite high values of correlation for some of the metrics.

Scalabrino et al. [SBV+17] conducted a study where they calculated correlations between 121 code-, documentation- and developer-related metrics and values for understandability gathered in an experiment with professional developers. They concluded that none of the investigated metrics could accurately represent code understandability. Even after repeating the study with increased sample size, the results did not change. They found however that, while still not fully capturing code understandability, combining metrics in models did improve their effectiveness [SBV+19]. Trockman et al. [TCM+18] reanalyzed the data set from Scalabrino et al. and put a renewed focus on combined metrics. They employed different statistical methods and found that code features had a small but significant correlation with understandability. In conclusion, they suggest that a useful metric for understandability could be created but more data would be needed to confirm that notion.

Feigenspan et al. [FALK11] measured the correlation of software measures with program comprehension from the data of maintenance tasks taken in an experiment with graduate students. The investigated variables for understandability were the correctness and response time of the performed tasks. They could not find a correlation between software measures and code comprehension.

## 1.2 Cognitive Complexity

Using different source code metrics to gather information about software quality has long been a staple of the software development process. Metrics such as lines of code (LOC) and McCabe's Cyclomatic Complexity (MCC) have found widespread usage in many development tools. In this chapter, a short introduction to cyclomatic complexity and the central metric that is the subject of this work, cognitive complexity, is presented.

### Cyclomatic Complexity

Introduced in 1976 by McCabe [McC76] as a complexity measure to control the complexity of Fortran programs, cyclomatic complexity has found use in a lot of software development tools and projects. Its calculation is done based on a program's control flow path. While it initially was designed for Fortran, it still has its use for modern programming languages. Listing 1.1 shows the calculation of MCC for a small Java method. The value is incremented for the `for` loop and the `if` and `else` conditional statements, resulting in a value of three.

```
1 public static void greeting(boolean isFriendly) {
2   for (counter = 0; i < 5; counter++) {                    // +1
3     if (isFriendly) {                                      // +1
4       System.out.println("Hello number " + counter + "! Good day.");
5     } else {                                               // +1
6       System.out.println("I'm in a bad mood today.");
7     }
8   }
9 }                                                          // 3 Total
```

**Listing 1.1:** Java greeting example with cyclomatic complexity calculation

As a general measure for complexity, MCC has received harsh criticism [She88; SSA13; Wey88]. Sheppard [She88] critiques the measure's theoretical underpinnings and claims that its usefulness as a predictor for software attributes has a lack of empirical evidence. Specifically, they claim that the measure is not based on objective evaluation but rather on intuitive notions of complexity. Furthermore, they assess that more often than not, LOC outperforms MCC for larger classes. They conclude that for a software metric to be seriously considered in the software engineering community, it requires a significant focus on the validation process.

Sarwar et al. [SSA13] criticize MCC based on what they call the nesting problem. They claim that nested loops should be seen as more complex than simpler constructs such as a nested conditional statement. Additionally, they emphasize the fact that the measure does differentiate by the number of iterations a loop has. For example, a single loop with two iterations is given the same value of complexity as a nested loop with 5000 iterations. They conclude that while MCC works fine for predicting testing efforts, it is insufficient as a basis for code comparison in terms of code efficiency.

MCC also does not fit the properties for evaluating syntactic software complexity measures proposed by Weyuker [Wey88]. One of the properties it violates is that there may only be a finite number of programs that have the same non-negative value of complexity. MCC does not differentiate between a program that does very little computing and a program that does a lot of computing as long as they have the same decision structure. Additional properties are violated due to MCC giving program bodies inherent complexities regardless of their context and its calculation of complexity for a program independent of the placement and interactions in relation to other statements.

#### Cognitive Complexity

In 2017, SonarSource[1] introduced cognitive complexity [Cam17] as a new metric for measuring understandability. Similar to cyclomatic complexity, cognitive complexity focuses on calculating the complexity of a given piece of code by focusing on the control flow structures. Cognitive complexity was specifically designed to address some of the issues raised by critics of the cyclomatic complexity metric. Moreover, they mention that cyclomatic complexity only manages to accurately measure testability, while cognitive complexity actually captures understandability. What follows is an explanation of the characteristics of cognitive complexity with snippets taken from the Java programming language. However, that does not mean that the metric was only designed for Java. In the white paper introducing cognitive complexity, hints and instructions are given for a multitude of programming languages, such as COBOL, JavaScript, and Python.

In general, cognitive complexity is built on three basic rules: First, ignore structures that allow multiple statements to be readably shorthanded into one. This means that there is no increment for a method declaration or null-coalescing operators like `??` in C# or PHP. Secondly, there is an increment for breaks in the linear flow. This includes structures like loops, conditionals, gotos, catch statements, sequences of logical operators and recursive methods. The last rule is that there is an increment for nested control flow structures. For example, a nested loop declaration would increase the complexity value by two, an additional structure nested within this loop would increase the value by three and so on [Cam18].

[1] https://www.sonarsource.com/

```
int sumOfPrimes(int max) {
  int total = 0;
  OUT: for (int i = 1; i <= max; ++i) {
    for (int j = 2; j < i; ++j) {
      if (i % j == 0) {
        continue OUT;
      }
    }
    total += i;
  }
return total;
}
```

**Listing 1.2:** Sum of primes example

```
String getWords(int number) {
  switch (number) {
    case 1:
      return "one";
    case 2:
      return "a couple";
    case 3:
      return "a few";
    default:
      return "lots";
  }
}
```

**Listing 1.3:** Get words example

Cognitive complexity specifically addresses some of the critiques of cyclomatic complexity as a software measure and attempts to improve upon them. As an example, compare the code snippets shown in Listing 1.2 and Listing 1.3. When using cyclomatic complexity, both code snippets would receive a value of four. Yet intuitively, the method in snippet Listing 1.3 seems to be more understandable. This exemplifies one of the critiques of cognitive complexity where control structures such as `switch` cases are evaluated the same way as nested `for` loops. In contrast, the cognitive complexity metric seems to more accurately represent the understandability of these snippets. Listing 1.3 has a cognitive complexity value of 7, which is seven times as high as for the snippet in Listing 1.2. While in this case, the metric appears to be right, for other programs this might not be the case.

This new metric is of special importance for the field of source code comprehension since it is specifically described as a measure of understandability [Cam18]. The initial paper included a small investigation of the developer reaction to the introduction of cognitive complexity in SonarCloud. They concluded that the metric had a 77% acceptance rate among developers. While the metric has already found success within the SonarSource software ecosystem under the SonarQube and SonarCloud programs, its usefulness as a software measure has yet to be tested in widespread studies. This work represents the first effort to empirically evaluate the validity of cognitive complexity as a measure of understandability.

# 2 Methods

In this chapter, the methods that were used in this research are presented, separated by the two main contributions of this work, the literature search and data analysis for cognitive complexity. The descriptions include the general methodological approach and details about how it was employed in practice.

## 2.1 Systematic Literature Search

As described in Chapter 1, there is a need for a comprehensive analysis of the merits of cognitive complexity as a measure for understandability. To this end, a systematic literature search was conducted to identify studies that measure source code understandability from a human developer's point of view. This work loosely follows the steps described by Kitchenham et al. in their guidelines for performing systematic literature reviews in software engineering [KC07]. The difference between the literature search conducted in this work and usual systematic literature reviews stems from the difference in the process of data extraction. Instead of aiming to create a summary of the topics discussed in the papers from the search, the goal of this search is to identify and extract the code snippets used in and the experimental data generated by the studies included in these papers. This means that while this literature search looks at the full text of the studies for additional context, it mainly focuses on identifying included data sets. The aim was to find code snippets for which the understandability was evaluated by humans. Finally, these understandability values were correlated with cognitive complexity of the snippets to gain knowledge on the metric's merit as a measure of source code understandability.

**Research Questions** With the literature search and correlation analysis conducted in this work, the following research questions were aimed to be answered:

**RQ1**: *Does cognitive complexity correlate with measures of understandability from existing studies?* The evaluation of cognitive complexity as a measure of understandability is the main goal of this work. Answering this question requires a comprehensive analysis with large sets of data. All data sets that were found through the literature search were separately inspected and prepared to be used in the analysis. The cognitive complexity values for code snippets were gathered and their correlation with the understandability variables from the experimental data was evaluated. Depending on the results of this analysis, an assessment was made about the validity of cognitive complexity for understandability.

**RQ2**: *What comprehension tasks are used in existing studies?* To better understand code comprehension and how it is measured, it is important to look at how other studies have done this in the past. By looking at which tasks the study subjects were asked to perform, an overview was created of what usually is considered as the proper way of gauging that somebody understood a piece of code.

**RQ3**: *What programming languages are most commonly used in understandability studies?* Which programming language to choose when conducting an experiment is always a question up to the researchers. What programming languages are the most popular in experiments about source code understandability is presented.

**RQ4**: *How has the field of code understandability developed over time?* Code comprehension as a field in the computer sciences has been around since the 80s. Over time, there have been changes to how experiments are conducted and what tools are at the disposal of researchers. To gain a better picture of this evolution, the popularity of the employed programming languages, understandability measures as well as the number of comprehension experiments in general are compared and their evolution over time is described.

While **RQ1** is the main research question to be answered by this thesis, additional research questions **RQ2** to **RQ4** were formulated to give the reader a better understanding of the nature of understandability studies and the results of this literature search in general. For **RQ2** to **RQ4**, the data from all papers which were identified as relevant was used as the necessary information could be found in their full text and no experimental data set was required. For **RQ1** specifically, only the literature where the experimental data was available could be used.

### 2.1.1 Search Strategy

In this section, the general search strategy employed in the literature is described. Figure 2.1 shows an overview of each of the steps of this strategy in chronological order.

**Search Terms** To fulfill the aim of this literature search, it was necessary to find adequate search terms to describe relevant studies. These terms had to be general enough to include understandability and all its synonyms but sufficiently specific to not include too many irrelevant studies. As mentioned in Section 1.1 there is a multitude of terms describing source code understandability and put into use by different researchers when conducting code comprehension experiments. To include all the relevant studies, some of these synonyms were used as part of the search string. The decision was made to also include studies that measure readability due to its close relation to understandability and the issue that some studies' definition of readability are indistinguishable from definitions of understandability. The general search string used was as follows:

General Search String

(code OR software OR program) AND (understandability OR comprehension OR readability OR complexity OR analyzability OR "cognitive load")

The first part of the string, code OR software OR program, was used to make sure that the results would relate to software engineering and program source code. This qualifier was then combined with understandability and other terms closely related or often used synonymously to it. The search string should bring about all studies on the subject of code understandability. From this point, a more fine-grained approach was used to filter and end up with the right studies.

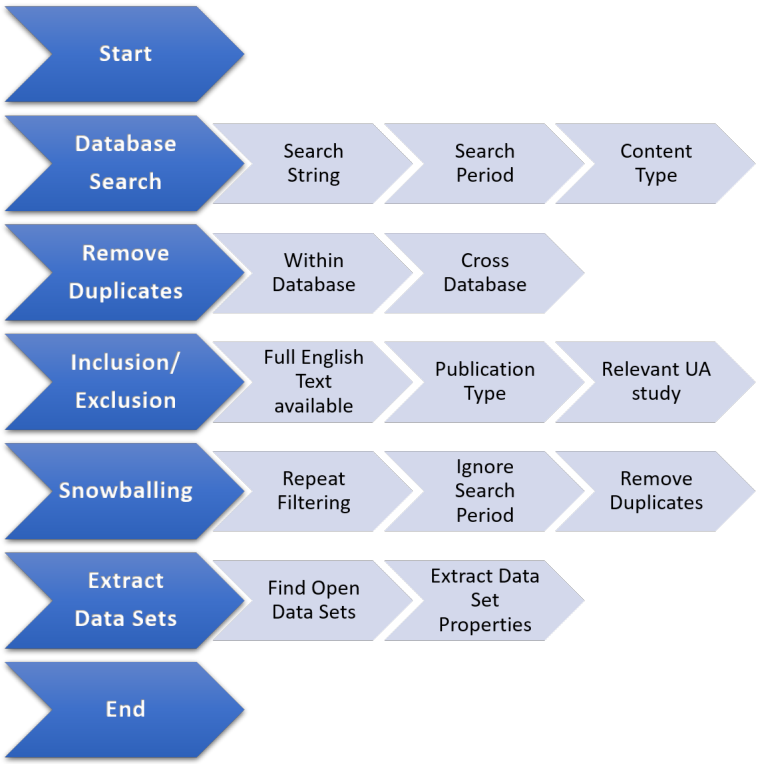


**Figure 2.1:** Summarized overview of the literature search process

**Data Sources** The aim of gathering a sufficient number of papers warranted the use of multiple online data sources. Table 2.1 shows all the sources that were used to search for literature.

| Data Source | URL |
|---|---|
| Google Scholar | http://scholar.google.com/ |
| ACM Digital Library | https://dl.acm.org/ |
| IEEE Xplore | https://ieeexplore.ieee.org/ |
| Elsevier ScienceDirect | https://www.sciencedirect.com/ |

**Table 2.1:** Data sources included in the literature search and their URLs

Google Scholar is a search engine provided by Google for searching scientific literature. It aggregates literature published by a wide variety of different publishers. Due to this nature as an aggregator it is likely that results in Google Scholar include duplicates from other publisher's search engines. This makes a check for duplicates especially important when using results from multiple different data sources.

The ACM Digital Library is an online library for papers published under publications from the Association for Computer Machinery (ACM). The focus of this literature lies on topics surrounding computing and information technology.

IEEE Xplore is an online database for literature published by the Institute of Electrical and Electronics Engineers (IEEE) and a small number of other select scientific publishing partners. Research topics range from computer science to electronics and electrical engineering.

Elsevier ScienceDirect is the online search engine for scientific research articles and books published by Elsevier. Compared to the other data sources used in this literature search, Elsevier ScienceDirect has less of a focus on the literature from engineering disciplines and also includes papers from other research areas such as social and health sciences.

**Search Period** All papers published between 2010 and July 2019, the time the literature search was conducted, were included in the search results. This was done to reduce the number of search results as well as to focus on generating results that more closely fit into the inclusion criteria of the search strategy. Because the main goal of this literature search was to find data sets from understandability studies that could be used in an analysis, a major focus was put on identifying studies with open data sets. Open-source research has been a trend for a few years, but it has truly only been gaining steam more recently. Due to this development, it made sense to limit the search period to more recent years, since the studies published then would be most likely to have openly published their data sets.

While this reduced the number of studies that had to be tested against the criteria, it also meant that some relevant studies might be excluded even though they would fit all other criteria. To mitigate this effect, an additional snowballing step was added to the search strategy that ignored the search period restriction described in Section 2.1.1 in more detail.

**Inclusion and Exclusion Criteria** A summary of the criteria used to filter the literature which was described in the previous sections is given in Table 2.2. The exact way these criteria were enforced for each data source is explained in more detail in Section 2.1.2.

| |
|---|
| **Inclusion criteria** |
| • Title contains the search string |
| • Published in a peer-reviewed journal/conference entry or workshop |
| • Published after 2010 |
| • Reports on an experiment measuring the understandability of code from a human point of view |
| **Exclusion criteria** |
| • Paper is not related to the field of software engineering |
| • Paper is a duplicate entry |
| • Paper is not available in English |
| • Full text is not available |

**Table 2.2:** Inclusion and exclusion criteria

**Snowballing** Kitchenham et al. [KC07] suggests searching for additional papers in the reference lists from relevant primary studies, a process commonly referred to as snowballing. New papers are extracted from the references given in the paper or papers that were previously marked as relevant. Here, a single iteration of backward snowballing was employed on the papers remaining after all filtering steps were applied as it is described by Wohlin [Woh14]. This was done to find more papers that were not included in the initial search results but are potentially still relevant. Each reference then was put through the same filtering steps of duplicate removal and inclusion/exclusion criteria, except for the time period cutoff. Special importance was put on this snowballing step due to the fact that the initial results restricted the results to the years 2010 and later. This way, the studies conducted before the time cutoff were still regarded in the final search results as long as they were mentioned in one of the identified studies.

**Data Extraction** At this point, it is assumed that all papers about relevant studies were found. The full text of each paper was then explored to identify whether they had a publicly published data set. Where this was not the case, but the study mentioned appeared to be of value to the analysis, the authors were contacted to request access to the code snippets and experimental data. All papers where no published data set could be found are then removed. From the remaining papers, the resulting data sets had to be extracted. For this study, the properties named in 2.3 of each study were stored in a spreadsheet.

| | |
|---|---|
| **Paper Reference Data** | title, year, author, publication type, country of origin, institution |
| **Study Properties** | participants, demographic, comprehension tasks |
| **Data Set Properties** | number of code snippets, programming language, analyzed code metrics, understandability measures |
| **Data** | the data set and software system or code snippets |

**Table 2.3:** Data points for extraction from literature search results

**Data Synthesis** The data synthesis step differed from the usual SLR process due to the nature of the search results. Instead of just using the full-text to review the study methodology or subjects, the actual data sets resulting from these studies were the main objective of this literature search. The data sets required an extensive data analysis to be conducted to prepare them for the usage with the cognitive complexity metric. The methodology of this analysis can be found in Section 2.2 and the results in Chapter 3.

### 2.1.2 Search Methods

Due to the differences in the data sources' search functionality as well as their behavior in handling exports, the search method differed for each source. This section describes the search process for each data source and the general methods for handling the data in detail.

**Google Scholar** The search string used for Google Scholar was as follows:

Google Scholar Search String

allintitle: (code OR software OR program) AND (understandability OR comprehension OR readability OR complexity OR analyzability OR "cognitive load")

The prefix “allintitle:” made the search title-only.

In accordance with the search strategy, the following inclusion/exclusion filtering was applied:

- **Include**: -
- **Exclude**: Citations, Patents

The time period for publishing was then set to 2010 to now.

Due to Google Scholar not offering an easy way to export the search results at the time of conducting the literature search, a workaround had to be used. Instead of applying a third-party solution, the “my library” function of Google Scholar was used. Each search entry was manually added to a personal library. Once all results had been added, the entries in the library could then be exported page by page with Google Scholar’s own export functionality. It is important to note that this solution has its limits. At the time of writing, Google Scholar only displays 100 pages of search results. This means that if one were to attempt to add the results of more than 100 pages to their personal library they would not be able to do so. Fortunately, the results for this literature search did not exceed these limits.

Because Google Scholar was the only data source that did not allow filtering by literature type besides citations and patents, an additional step of filtering was required for its search results. This means that during the manual filtering stage of this literature search, the results obtained from Google Scholar additionally hat to be checked for whether they were papers published in peer-reviewed journals.

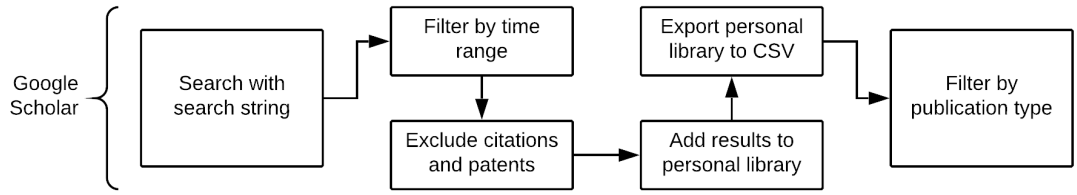


**Figure 2.2:** Google Scholar search process

**ACM Digital Library** The search string used for the ACM Digital Library was as follows:

ACM Digital Library Search String

acmdlTitle:((code OR software OR program) AND (understandability OR comprehension OR readability OR complexity OR analyzability OR "cognitive load"))

The prefix “acmdlTitle:” made the search title-only. Additional parentheses after the prefix applied it to all search terms.

The ACM Digital Library allowed the results to be filtered by ACM publication type. In accordance with the search strategy, the following inclusion/exclusion filtering was applied:

- **Include**: Proceeding, Journal
- **Exclude**: Newsletter, Magazine, Book

The time period for publishing was then set to 2010 to now. All search results were then exported using on-site functionality. The entries for each ACM Publication had to be exported separately, resulting in multiple CSV files that were then combined.

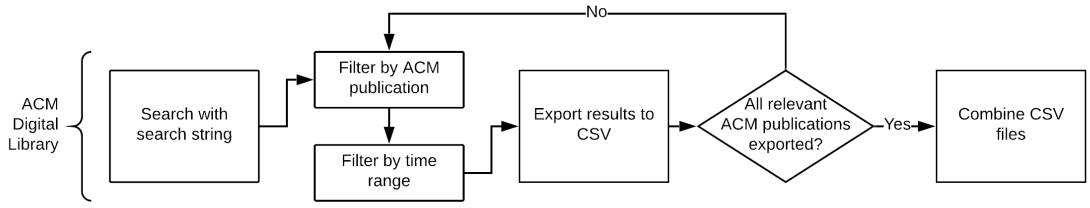


**Figure 2.3:** ACM Digital Library search process

**IEEE Xplore** The search string used for IEEE Xplore was as follows:

IEEE Xplore Search String

```
("Document Title":code OR "Document Title":software OR "Document Title":program)
AND ("Document Title":understandability OR "Document Title":comprehension OR
"Document Title":readability OR "Document Title":complexity OR
"Document Title":analyzability OR "Document Title":"cognitive load")
```

The prefix “"Document Title":” made the search title-only. To apply this prefix to all search terms, it had to be added before each individual term.

IEEE Xplore allowed the results to be filtered by content type. In accordance with the search strategy, the following inclusion/exclusion filtering was applied:

- **Include**: Conferences, Journals & Magazines, Early Access Articles
- **Exclude**: Books, Courses, Standards

The time period for publishing was then set to 2010 to now. All search results were then exported using on-site functionality.

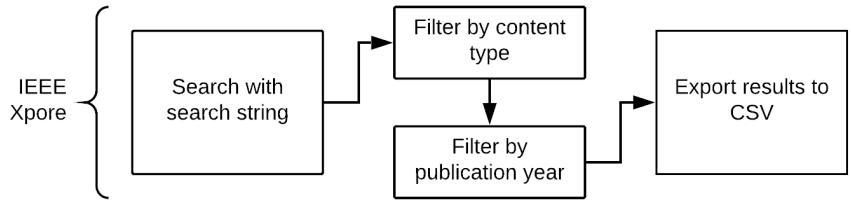


**Figure 2.4:** IEEE Xplore search process

**Elsevier ScienceDirect** The search string used for Elsevier ScienceDirect was as follows:

Elsevier ScienceDirect Search String

(code OR software OR program) AND (understandability OR comprehension OR readability OR complexity OR analyzability OR "cognitive load")

To employ a title-only search, the advanced search function had to be used. The search string was entered in the “Title:” field in the advanced search.

Elsevier ScienceDirect allowed the results to be filtered by article type. In accordance with the search strategy, the following inclusion/exclusion filtering was applied:

- **Include**: Review articles, Research articles, Data articles
- **Exclude**: Encyclopedia, Book chapters, Conference abstracts, Book reviews, Case reports, Conference info, Correspondence, Discussion, Editorials, Errata, Examinations, Mini reviews, News, Patent reports, Practice guidelines, Product reviews, Short communications, Software publications, Video articles, Other

The time period for publishing was then set to 2010 to now. All search results were then exported using on-site functionality. ScienceDirect did not allow export to a CSV file like the other data sources. Instead, they were exported to the BibTex format and then converted to a CSV file with JabRef[1].

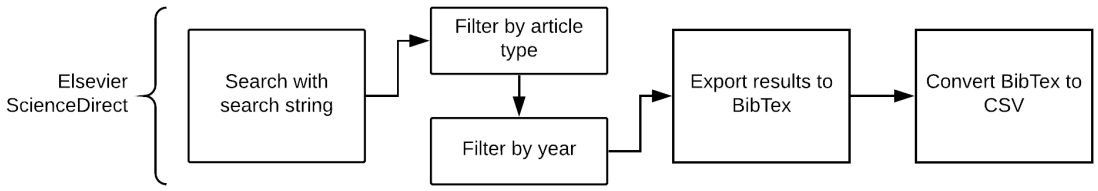


**Figure 2.5:** Elsevier ScienceDirect search process

[1] http://www.jabref.org/

### Filtering Process After Exporting

Figure 2.6 shows the filtering methodology employed in this literature search. This includes all steps once the search results had been exported from the data sources as well as which data artifacts were used and what software was employed.

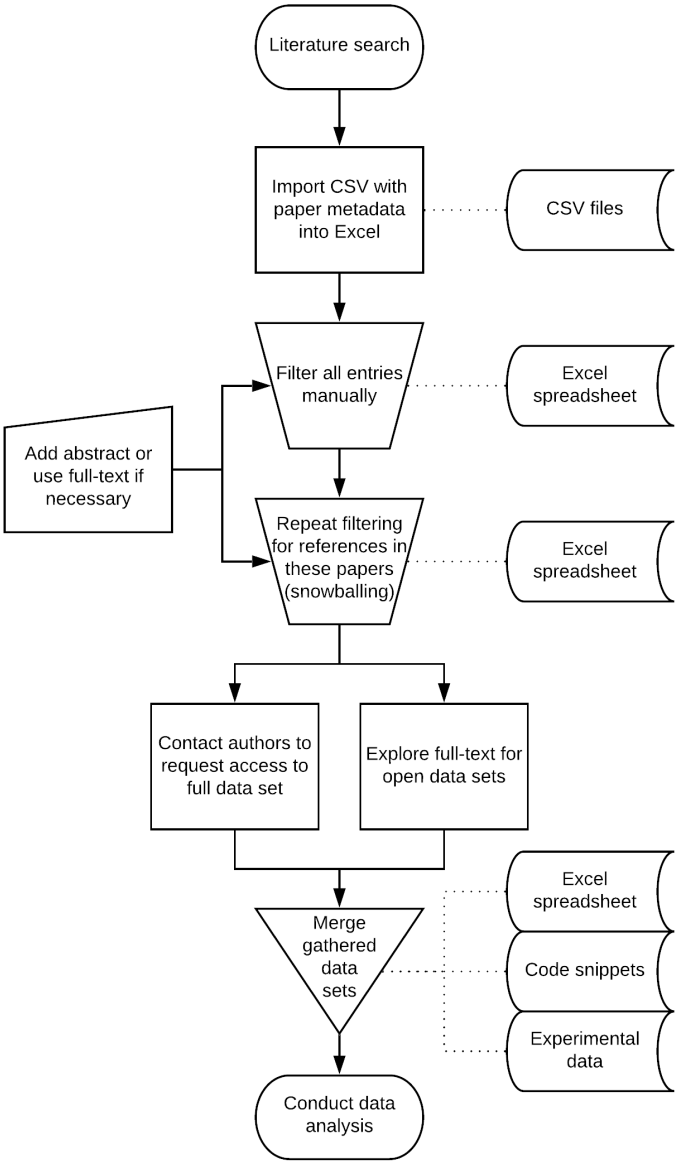


**Figure 2.6:** Flowchart with the general search process steps and the corresponding data artifacts

Once the reference entries were exported from each data source, they had to be combined and converted into a human-readable format. Using Microsoft Excel [2], each CSV file was imported and saved as an Excel spreadsheet. This put each metadata field from the CSV files in its own column resulting in human-readable spreadsheets and allowing for advanced manipulation with Excel formulas and operations.

To check for duplicates, the Excel feature of conditional formatting was used in order to highlight entries with an identical title. These highlighted entries were then compared on the basis of their metadata to decide whether an actual duplicate was identified. If the duplicate was deemed to be accurate, the corresponding entry was removed from the spreadsheet. Each of the remaining entries was then looked at one-by-one manually to evaluate whether it fits the inclusion and exclusion criteria described in the search strategy. Their suitability was checked by looking at the title, abstract and other metadata. Where the title and abstract were not sufficient to determine suitability, the full text had to be consulted. The snowballing step was done by simply copying the entries in the reference section of each paper into another spreadsheet. These new entries were put under the same duplicate removal and manual filtering steps, except the time period criterion.

As a final selection criterion, the full text of each paper was examined to locate any openly published data. Additionally, the papers where no data set was published, but the work described seemed like a good fit for this study, the authors were contacted to request access to the data set. This included all studies that were published since 2000, where it was expected that the authors still had access to the experimental data and where at least ten code snippets were evaluated to ensure a higher chance of significant correlation analysis results. The final results of the search and filtering process were a spreadsheet with metadata entries for each paper, the experimental data and the code snippets as provided by the studies.

### 2.1.3 Search Results

In this section, the results of the previously described literature search are presented. First, the number of papers excluded in each filtering step is visualized. Then the four research questions are addressed with the results from the search. At last, an overview of the papers that managed to pass the final selection criteria and were used in the data analysis is presented.

Figure 2.7 shows the data flow of papers that were analyzed during the filtering in the literature search with the number of excluded literature annotated for each step. Similarly, Figure 2.8 shows this flow during the subsequent snowballing procedure.

Without any filters applied, most literature was found through Google Scholar, with the ACM Digital Library and IEEE Xplore closely following. Only Elsevier ScienceDirect showed a significantly smaller number of relevant results, only making up about 6% of the total amount. In total, out of the initial 7,068 papers, only 28 were deemed suitable for this study. The biggest filter was the restriction on the time period for publishing. Even with all the automatic filtering steps removing about 4,711 entries, the number of papers that had to be manually checked for suitability was still significant (2,357).

[2] https://products.office.com/en-us/excel

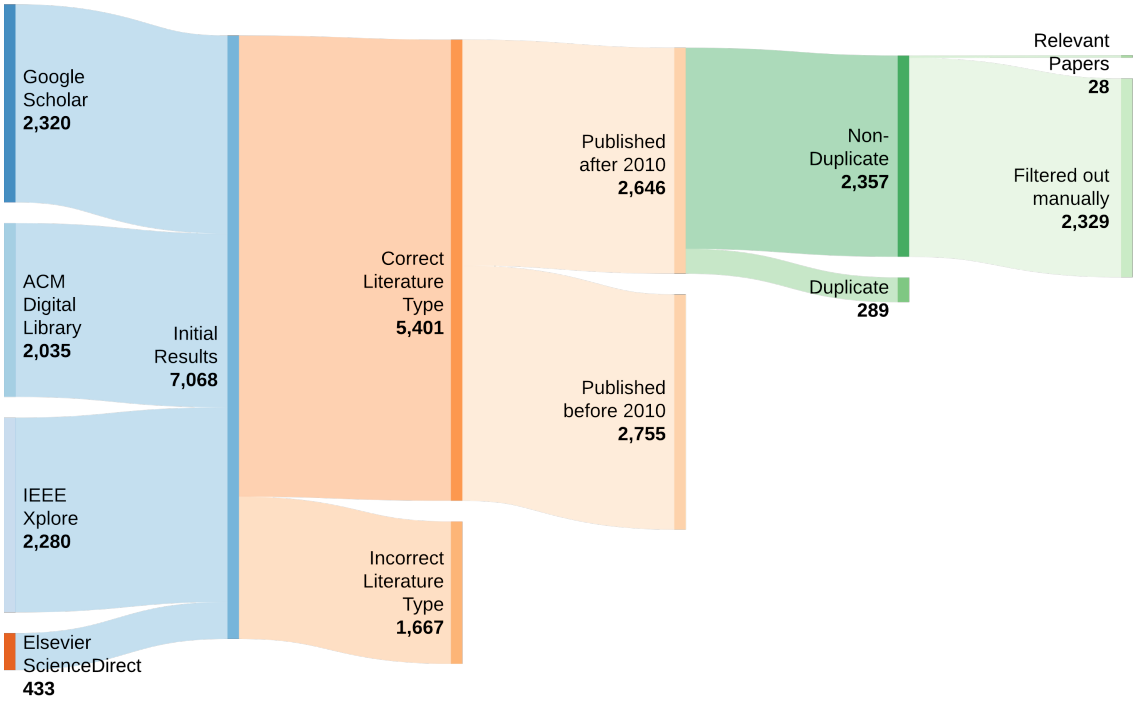


**Figure 2.7:** Sankey diagram showing the results of the literature search

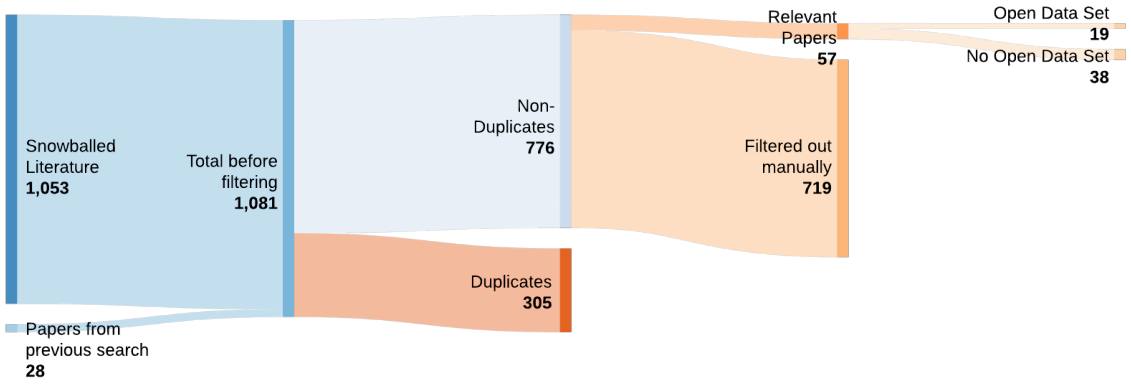


**Figure 2.8:** Sankey diagram showing the results of snowballing

Snowballing proved to be an effective tool to widen the focus of the search, raising the number of relevant papers from 28 to 57. The majority of the new papers, 20 out of 29, were published before 2010. A possible explanation for this could be the restriction of the publishing period being lifted for snowballing. Of the final 57 relevant papers, only 19 offered an openly accessible data set.

### Research Questions

**RQ1** *Does cognitive complexity correlate with understandability measures from existing studies?* To answer this question, a comprehensive correlation analysis was conducted. The results of this analysis can be found in Chapter 3.

**RQ2** *What comprehension tasks are used in existing studies?* The different types of comprehension tasks used to validate whether study participants understood the software in question can be seen in Figure 2.9. Each occurrence of a task type in a study is counted as a data point for the chart. Since some studies used multiple task types at once the total number of entries in the diagram is higher than the number of papers. Most studies included questions to validate comprehension such as finding the correct output or describing the purpose of a program. Another popular way of determining the comprehensibility of a program was to ask subjects their opinion on how difficult to understand the program is. The most common way of doing this was to use a Likert scale [Lik32] or a NASA-TLX [HS88] questionnaire. Some also included maintenance tasks, for instance locating and sometimes fixing a syntactic or semantic defect in a program. Few made use of more advanced tasks like refactoring, extending or modifying larger parts of code. A small number of studies also employed tasks similar to a cloze test [Tay53], in which subjects were asked to fill out missing parts of a program. Exercises involving testing the memory of the comprehension of subjects were rarely used. In these tasks, they were shown the complete code snippet for some amount of time and then were asked to recall some elements such as identifiers or parameter names.

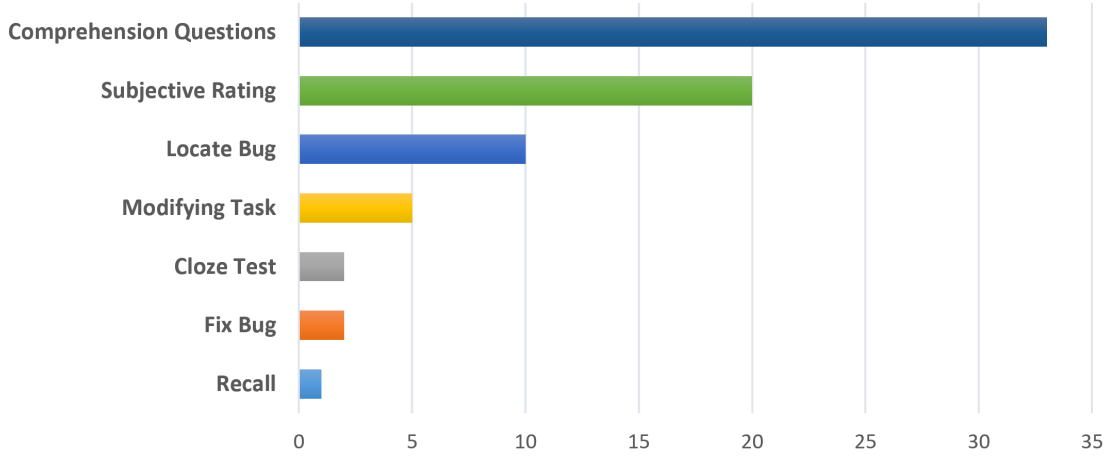


**Figure 2.9:** Pie chart showing the type of comprehension task used in the studies

**RQ3** *What programming languages are most commonly used in understandability studies?* Figure 2.10 shows the number of times each programming language was used in one of the studies. The most popular programming language that was Java, followed by C/C++. Programming languages enjoying moderate popularity, about 2 to 5 uses, were FORTRAN, Python, Pascal, C#, and Scala. Languages depicted as “Other” in Figure 2.10 included ALGOL 68, JavaScript, Trygve, AspectJ, CUDA, Visual Basic, Modula-2, and PL/1. All of these languages were only used once.

**RQ4** *How has the field of code understandability developed over time?* The first relevant study was published in 1979. Figure 2.11 shows the number of published understandability studies by year since then. Over time, the popularity of understandability studies was relatively stable, with a maximum of two published studies in a year until the start of the 2010s. From the year 2011, there seems to be a spike in the number of relevant studies. The largest number of relevant studies were published in 2017, eight in total.

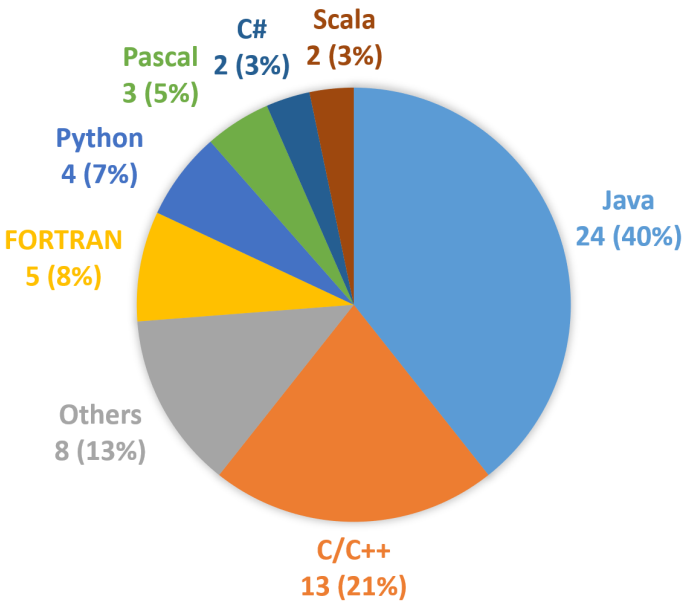


**Figure 2.10:** Pie chart with the programming languages used in the studies

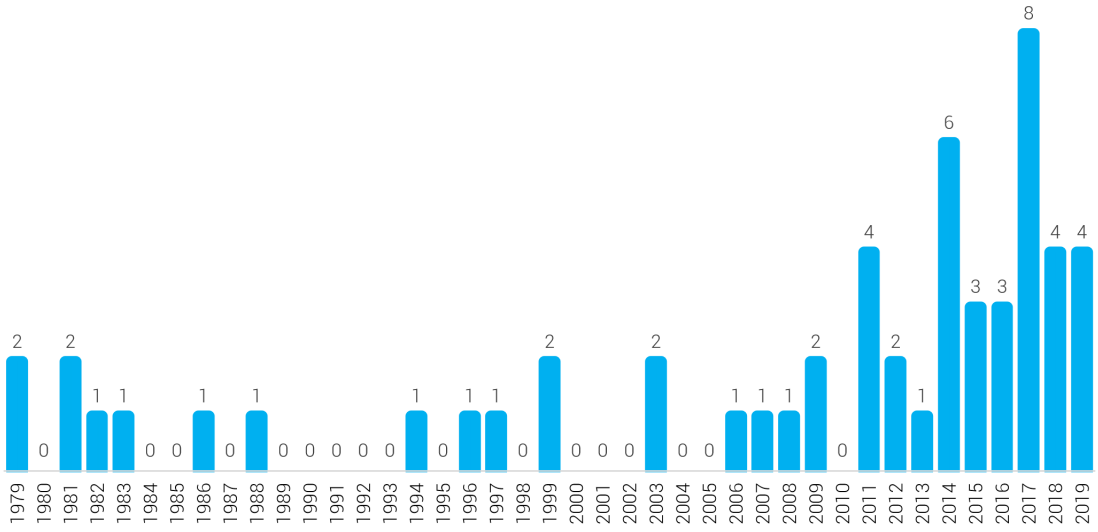


**Figure 2.11:** Histogram showing the number of published understandability studies by year

In addition to the general rise in the popularity of understandability studies, there was also a change in the properties of these studies that could be observed. Over time, different programming languages and measures for understandability were the focus of these studies. While measures such as response time, subjective ratings and correctness of comprehension tasks seemed to show a steady usage since the first conducted experiments, physiological measures such as EEG, EDA, fMRI, fNIRS, heart rate monitoring and eye-tracking are a newer trend that only showed up from 2010 and later.

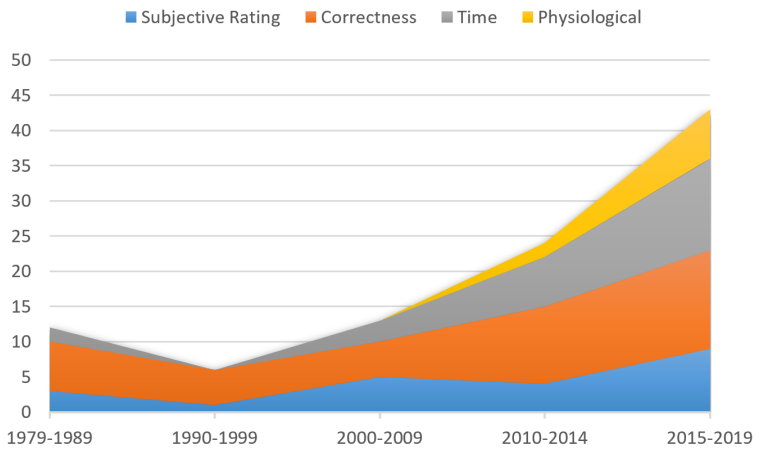


**Figure 2.12:** Area chart showing the types of measures used in studies over time

Similar to the ways understandability is measured, the programming languages used in these studies have also changed over time. While older programming languages such as FORTRAN or Pascal were used in the pioneering experiments in the field, modern studies opted to use more contemporary languages like Java or C++. Partly this can be explained because then not all languages existed but moreover, it is also in line with the general popularity of these programming languages in academic and professional contexts [RCM15; YDM+05].

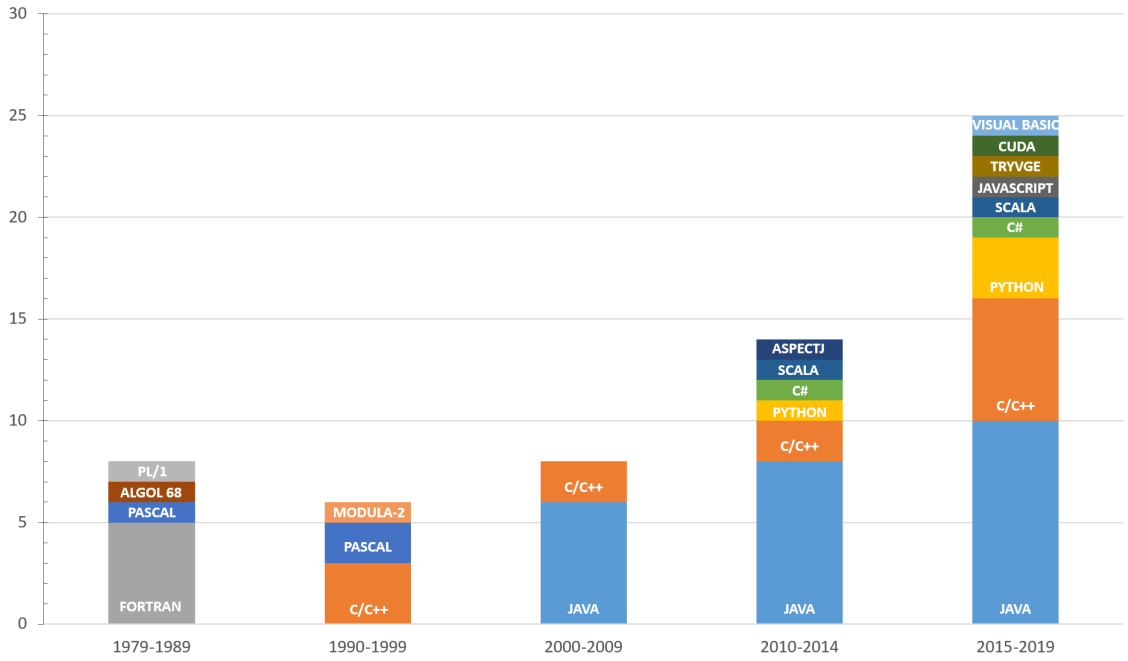


**Figure 2.13:** Histogram showing the programming languages used in studies over time

#### Data Extraction

The final set of papers where a public data set could be found in the full text are shown in Table 2.4.

| Reference | Title | Year |
|---|---|---|
| [PSA+18] | A Look into Programmers' Heads | 2018 |
| [BW08] | A metric for software readability | 2008 |
| [FGN+19] | A Replication Study on Code Comprehension and Expertise using Lightweight Biometric Sensors | 2019 |
| [DHOH03] | An empirical investigation of the influence of a type of side effects on program comprehension | 2003 |
| [AKGA11] | An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension | 2011 |
| [SAPM14] | An Empirical Study on Program Comprehension with Reactive Programming | 2014 |
| [TFSL14] | An Eye-tracking Study Assessing the Comprehension of C++ and Python Source Code | 2014 |
| [SBV+19] | Automatically Assessing Code Understandability | 2019 |
| [SBV+17] | Automatically assessing code understandability: how far are we? | 2017 |
| [FSW17] | Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise | 2017 |
| [FALK11] | Exploring Software Measures to Assess Program Comprehension | 2011 |
| [BW10] | Learning a Metric for Code Readability | 2009 |
| [HSH17] | Shorter identifier names take longer to comprehend | 2017 |
| [AWF17] | Syntax, Predicates, Idioms - What Really Affects Code Complexity? | 2017 |
| [FMAA18] | The effect of poor source code lexicon and readability on developers' cognitive load | 2018 |
| [BP16] | The Role of Method Chains and Comments in Software Readability and Comprehension—An Experiment | 2016 |
| [GIY+17] | Understanding Misunderstandings in Source Code | 2017 |
| [SKA+14] | Understanding understanding source code with functional magnetic resonance imaging | 2014 |
| [FBM+14] | Using Psycho-physiological Measures to Assess Task Difficulty in Software Development | 2014 |

**Table 2.4:** Relevant studies as identified by the literature search

Once all relevant studies were identified, the data extraction process could be started. This meant that the full text of each study was examined and all necessary data points were extracted. The final list of data sets used to calculate correlations and their properties can be seen in Table 2.5. The data set identifier (DID) represents a unique identifier for each of the data sets and will be used continuing from this point. The table also shows the programming language (PL), the number of code snippets (SNo), and the number of participants (PNo) for each data set. Finally, it shows which measures were taken as proxies for understandability in each data set.

| DID | Reference | PL | SNo | PNo | Understandability measures |
|---|---|---|---|---|---|
| 1A | [PSA+18] | Java | 23 | 41 | time, correctness, ratings |
| 1B | [PSA+18] | Java | 12 | 16 | time, physiological |
| 1C | [PSA+18] | Java | 23 | 57 | time |
| 2 | [BW10] | Java | 100 | 121 | ratings |
| 3A | [DHOH03] | C/C++ | 12 | 33 | time, correctness |
| 3B | [DHOH03] | C/C++ | 8 | 18 | time, correctness |
| 3C | [DHOH03] | C/C++ | 20 | 33 | time, correctness |
| 4 | [SAPM14] | Scala | 20 | 38 | time, correctness |
| 5 | [SBV+19] | Java | 50 | 63 | time, correctness, ratings |
| 6 | [FALK11] | Java | 6 | 21 | time, correctness, ratings |
| 7 | [HSH17] | C# | 6 | 72 | time |
| 8 | [AWF17] | JavaScript | 40 | 222 | time |
| 9 | [BP16] | Java | 30 | 259 | time, correctness, ratings |
| 10A | [GIY+17] | C/C++ | 126 | 48 | time, correctness |
| 10B | [GIY+17] | C/C++ | 8 | 22 | time, correctness |

**Table 2.5:** Data sets extracted from relevant studies

The number of data sets does not exactly match the number of relevant papers from the literature search as some papers included data sets from multiple studies, while for others, the same study was reported on in more than one paper. Both papers by Siegmund et al. [PSA+18; SKA+14] referred to the same set of experiments. Their data set was split into the first pilot study (1A), their study inside an fMRI scanner (1B), and both data sets combined (1C). Data from their second pilot study had to be discarded because it could not be determined which comprehension task belonged to which code snippet. Both identified papers by Buse et al. [BW08; BW10] referred to the same data set (2). The data set from Fucci et al. [FGN+19] unfortunately had to be discarded since the final experimental data was aggregated per subject instead of being split between code snippets. To make the data usable, all R scripts would have had to be rewritten which was deemed to be too much effort for this work. The data set from the study by Dolado et al. [DHOH03] was split into three: The simple tasks (3A), complex tasks (3B), and both tasks combined (3C).

The data from the eye-tracking study by Turner et al. [TFSL14] and the fMRI study by Floyd et al. [FSW17] could not be retrieved because both were no longer accessible. The study by Scalabrino et al. [SBV+19] included the data points from their original study [SBV+17]. The AspectJ code snippets from the study by Feigenspan et al. [FALK11] could not be automatically analyzed, so they were discarded. The replication packages from the studies by Fakhoury et al. [FMAA18], Weimer et al. [BW10], and Fritz et al. [FBM+14] included the documentation and code snippets needed to repeat their experiment but not the experimental data, so they could not be used for the correlation analysis. The data from Gopstein et al. [GIY+17] was split between the initial snippet study (10A) and their atom impact study (10B).

## 2.2 Data Analysis

In this section, the methodology of the second main contribution of this work is described. Moreover, the methods are explained that were employed to correlate the cognitive complexity of the code snippets with the measures from the experimental data identified by the literature search. The section is separated into two parts. The first part contains a short description of the steps that were used to prepare the code snippets and experimental data for the analysis. The second part describes how the measures from different subjects were used to create a proxy for understandability and which formulas were used to calculate the values for correlation.

### 2.2.1 Preparation

The values of cognitive complexity for each of the source code snippets were gathered with SonarQube[3]. SonarQube allows static code analysis for a wide variety of different programming languages, including all languages used in this work. The process did differ from study to study, depending on the language used and the way the code snippets were provided. This section details how the original snippets had to be altered so SonarQube could be used.

#### Code Snippets

Code snippets written in **Java** were built with Gradle[4] and analyzed with the respective SonarQube plugin. This meant that all snippets had to be compilable, meaning free of syntax errors and dependency issues. Unfortunately, most of the snippets provided by existing studies did not meet these criteria. They had to be altered accordingly to allow compilation without changing the cognitive complexity values. The most common changes were modifying method names if duplicates existed, wrapping code snippets in methods or classes, and adding imports for dependencies.

Again, the numbers of the data sets are defined in Table 2.5. For example, data sets 1 and 9 required additional `import` statements to be added to the snippets such as `import java.util.Arrays` and `import java.util.Collection`. More complex classes required additional libraries to be included in the project build, which was the case for data sets 5, 6, and 9. Where the right libraries could not be identified, the dependencies were just replaced with so-called dummies. These classes, methods, and fields just mirrored the same structure but were without class or method body, i.e. with no actual functionality. This was done for data sets 2, 5, and 9.

The code snippets from data set 2 required a lot of modification. In this case, the snippets did not contain complete methods but instead included incomplete code structures. These had to be fixed to make the snippets syntactically correct and allow subsequent compilation and analysis. Listing 2.1 shows an example of one of the incomplete code snippets from data set 2. The `if` conditional was completed, an additional `return` statement was added, and the snippet was wrapped in a method. The modified snippet can be seen in Listing 2.2. Both examples have the same value for cognitive complexity but the latter can be analyzed with SonarQube.

[3] https://www.sonarqube.org/
[4] https://gradle.org/

```
if (actionList.size() == 1) {
  ActionMenu menu = actionList.get(0);

  if (menu.getSubItems().length == 0) {
    return null;
  }

  if (menu.getSubItems().length == 1) {
    Action action = menu.getSubItems()[0].
getAction();
```

**Listing 2.1:** Unaltered code snippet

```
public static Object s2() {
  if (actionList.size() == 1) {
    ActionMenu menu = actionList.get(0);

    if (menu.getSubItems().length == 0) {
      return null;
    }

    if (menu.getSubItems().length == 1) {
      Action action = menu.getSubItems()[0].
getAction();
    }
  }
  return null;
}
```

**Listing 2.2:** Modified code snippet

**C** and **C++** projects were analyzed with the SonarQube C++ community plugin[5] since the community edition of SonarQube did not allow analysis for these languages. Some snippets in data set 3 had syntax errors such as a missing `;` or a space between `= =` which had to be fixed. The **Scala** and **JavaScript** code snippets could be analyzed with the Sonar scanner without modification. The **C#** snippets from data set 7 were wrapped in classes and dependencies were added with `using` when needed. To calculate cognitive complexity, the semantic error version of each snippet was used since the syntactic error versions could not be compiled. The cognitive complexity values of both are the same since the syntactic errors did not change any control flow structures.

Not all studies provided the code snippets in source code files. For data set 5, only the URLs to the corresponding projects that included the code snippets were provided. Since the compilation of all of these projects proved to be difficult, just the methods themselves were extracted and added to new classes. Additionally, the code snippets from data sets 1, 3, 4, and 9 were provided in PDF files, so they first had to be copied into the appropriate source code files. For data set 10, the source code was provided on the project website[6] in text form. These snippets were also copied into source code files.

### Experimental Data

For the experimental data, small adjustments had to be made to some data sets so that the proxy variables of understandability could be used to calculate the correlations. For data set 1B, the code snippet used for each comprehension task was not explicitly named. Fortunately, the right snippet could still be determined by the description of the functionality given by the respondents in each task. We combined the time variables of data sets 1A and 1B into a new data set 1C since both studies used the same code snippets. While the initial pilot study 1A captured time in seconds, the study in the fMRI scanner (1B) used milliseconds. To make the combination of the two work, before adding the individual data points to the new data set, all values from the second study were also converted to seconds.

[5] https://github.com/SonarOpenCommunity/sonar-cxx/
[6] https://atomsofconfusion.com/

The data from data set 3 was provided in `.sav` files for IBM's SPSS software. Listing 2.3 shows the script written in R that was used to convert the files into CSV sheets. These could then be imported into Excel and subsequently be used for correlation calculations. The correctness values for data set 6 were not included in the experimental data. They instead were found in the actual full-text of the paper itself.

```
1 library(foreign)
2
3 mydata = read.spss("C:\\source-path\\source-name.sav",to.data.frame=TRUE)
4
5 write.table(mydata,"C:\\target-path\\target-name.txt")
6
7 getwd()
```

**Listing 2.3:** SAV data conversion using R

### 2.2.2 Correlation

The measures for understandability differed from study to study and each study could provide one or multiple measures for understandability. Section 1.1.2 explains in detail how understandability is measured and how these variables come to be. For this study, each measure was analyzed separately.

Additionally, for each of the code snippets, multiple measurements from different subjects were taken for each of the measures. These measurements were aggregated and used to calculate the correlation. Specifically, for each of the code snippets, the mean was taken from individual values measured for the subjects and used as a proxy for the understandability of the snippet. An example of this can be seen in Figure 2.14. It shows the distribution of individual data points for the measurements from the subjects for five code snippets with different cognitive complexity values. Additionally, it shows the aggregated value or the mean value in red.

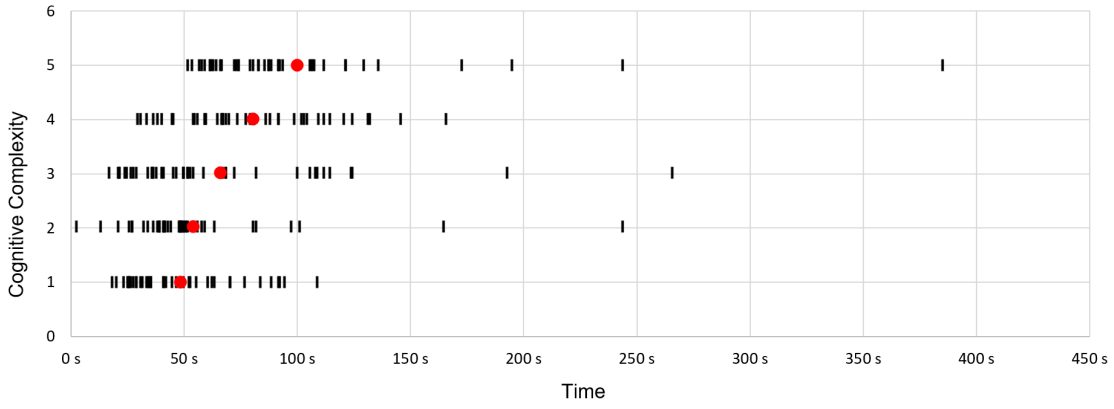


**Figure 2.14:** Point chart showing the aggregation of understandability values

The cognitive complexity value and the corresponding aggregated values for the understandability of a code snippet form the data points for the correlation analysis. Formally, the independent variable $X$ was the cognitive complexity value of a code snippet and the dependent variable $Y$ was the aggregated understandability value of a code snippet. To calculate Pearson's correlation coefficient the formula in Equation (2.1) was used.

$$\rho_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y} \tag{2.1}$$

where

cov is the covariance,
$\sigma_X$ is the standard deviation of $X$, and
$\sigma_Y$ is the standard deviation of $Y$.

Additionally, the Spearman's rank correlation coefficient was employed as well since it is possible that not all proxy variables completely fulfill each of the requirements of Pearson's correlation coefficient. The formula used for Spearman's correlation coefficient can be found in Equation (2.2).

$$r_s = \rho_{\mathrm{rg}_X,\mathrm{rg}_Y} = \frac{\mathrm{cov}(\mathrm{rg}_X,\mathrm{rg}_Y)}{\sigma_{\mathrm{rg}_X}\sigma_{\mathrm{rg}_Y}} \tag{2.2}$$

where

| | |
|---|---|
| $r_s$ | is the Spearman correlation coefficient, |
| $\mathrm{rg}_X$ , $\mathrm{rg}_Y$ | is the rank of the variables $X$ and $Y$, |
| $\mathrm{cov}(\mathrm{rg}_X,\mathrm{rg}_Y)$ | is the covariance of the rank variables, and |
| $\sigma_{\mathrm{rg}_X}$ , $\sigma_{\mathrm{rg}_Y}$ | is the standard deviation of the rank variables. |

# 3 Results

In this chapter, the results of the correlation analysis between cognitive complexity and understandability measures from existing studies are presented. This includes the values for correlation between the independent and dependent variables as well as a visual representation and a short explanation of differences between the variables, where needed. Table 3.1 shows the dispersion, range, and central tendency of cognitive complexity values for the code snippets of each data set.

| DID | Min | Max | Median | Mean | Standard Deviation | Variance |
|---|---|---|---|---|---|---|
| 1A | 0 | 9 | 3 | 3.26 | 2.43 | 5.93 |
| 1B | 0 | 6 | 2 | 2.50 | 1.88 | 3.55 |
| 1C | 0 | 9 | 3 | 3.26 | 2.43 | 5.93 |
| 2 | 0 | 10 | 1 | 1.34 | 1.74 | 3.01 |
| 3A | 0 | 3 | 2 | 1.33 | 1.23 | 1.52 |
| 3B | 1 | 8 | 4.5 | 4.38 | 2.50 | 6.27 |
| 3C | 0 | 9 | 2 | 2.55 | 2.35 | 5.52 |
| 4 | 0 | 7 | 2 | 2.55 | 1.82 | 3.31 |
| 5 | 0 | 46 | 7.5 | 8.56 | 8.53 | 72.74 |
| 6 | 0 | 59 | 5 | 22.17 | 28.6 | 817.77 |
| 7 | 1 | 4 | 2 | 2.17 | 1.33 | 1.77 |
| 8 | 1 | 23 | 5 | 6.50 | 5.17 | 26.72 |
| 9 | 0 | 16 | 4 | 8.00 | 6.81 | 46.34 |
| 10A | 0 | 8 | 1 | 1.17 | 1.40 | 1.96 |
| 10B | 1 | 47 | 7 | 12.5 | 15.41 | 237.43 |

**Table 3.1:** Descriptive measures of cognitive complexity for each of the data sets

Table 3.2 to Table 3.6 show the values for both Spearman's and Pearson's correlation coefficient, their significance through p-values and the number $n$ of code snippets for each of the variables measured in the studies.

One has to note, however, that while a positive correlation might indicate that cognitive complexity is a correct measure of understandability in some cases, this is not a general rule. For example, a positive correlation between cognitive complexity and the time taken to complete comprehension tasks would be expected, meaning that a less understandable snippet would take more time to be comprehended. But between cognitive complexity and correctness, a negative correlation would be expected since a less understandable code snippet should result in a smaller number of correct answers. These expectations are based on the assumption that the hypothesis stated in **RQ1** can be confirmed. To make the tables easier to comprehend, correlation values higher than $0.300$ or lower than $-0.300$ are marked in green if they support this assumption and in red if they contradict it.

To additionally make the interpretation of the correlation values easier, they are plotted in Figure 3.1 to Figure 3.10. Here, the x-axis represents cognitive complexity and the y-axis represents the inverse of understandability. This means that a line with a positive slope indicates a correlation as expected, that a snippet with a higher cognitive complexity would be less understandable. The green line represents a perfect positive correlation with a slope of $1$ and the red line is its reciprocal, a perfect negative correlation of $-1$. Lines closer to the x-axis represent no correlation. Finally, the color of each line indicates the significance of the plotted line. Lines where the calculated correlation value was determined to be insignificant, meaning that the p-value was higher than $0.05$ are shown in a gray tone. Lines for significant values are shown in black.

The slope values taken from the correlation coefficients were normalized to make them comparable. This meant that where a negative correlation was expected (e.g. correctness) the value was inverted, i.e. multiplied by $-1$. Specifically, this meant that a line showing a positive correlation close to a slope of $1$ meant that the correlation is as expected. The following sections include an explanation for each of the variables where this inversion was done.

## 3.1 Time

The variables for time are differentiated between the time taken to read and understand the code snippet (R), the time to complete the comprehension task (C), or both combined (T).

| DID | Variable | n | Pearson's r | p-value | Spearman's $\rho$ | p-value |
|---|---|---|---|---|---|---|
| 1A | Time (T) | 23 | 0.726 | <0.001* | 0.630 | 0.001* |
| 1B | Time (T) | 12 | -0.212 | 0.509 | 0.014 | 0.966 |
| 3A | Time (T) | 12 | 0.878 | <0.001* | 0.928 | <0.001* |
| 3B | Time (T) | 8 | 0.918 | 0.001* | 0.982 | <0.001* |
| 3C | Time (T) | 20 | 0.654 | 0.002* | 0.661 | 0.002* |
| 4 | Time (T) | 20 | 0.717 | <0.001* | 0.712 | <0.001* |
| 4 | Time (T) | 20 | 0.497 | 0.026* | 0.567 | 0.009* |
| 5 | Time (R) | 50 | 0.499 | <0.001* | 0.443 | 0.001* |
| 7 | Time (R) | 6 | 0.151 | 0.775 | 0.014 | 0.979 |
| 7 | Time (C) | 6 | -0.234 | 0.655 | 0.014 | 0.979 |
| 9 | Time (R) | 30 | 0.101 | 0.596 | 0.603 | <0.001* |
| 9 | Time (C) | 30 | 0.389 | 0.034* | 0.272 | 0.146 |
| 10A | Time (T) | 126 | 0.263 | 0.003* | 0.281 | 0.001* |
| 10B | Time (T) | 8 | 0.814 | 0.014* | 0.923 | 0.001* |

**Table 3.2:** Raw correlation results for comprehension time. (*p<.05)

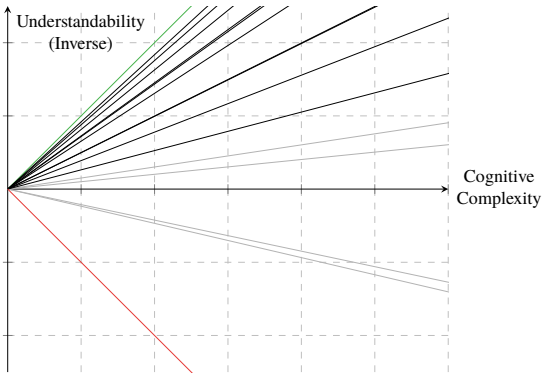


**Figure 3.1:** Interpreted Pearson correlation for time

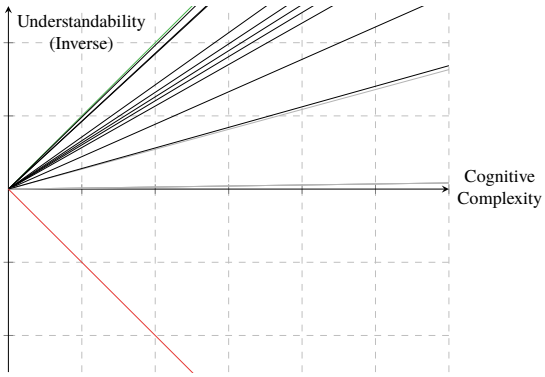


**Figure 3.2:** Interpreted Spearman correlation for time

**Black** lines represent significant values (p<0.05) and **gray** lines represent insignificant values (p>0.05). Again, these lines do not represent the values from Table 3.3 without change but rather the interpreted values with regards to **RQ1**. This means that lines with a positive slope show that a higher cognitive complexity means lower understandability.

Specifically, the correlation values of all time variables were used as-is since a positive correlation would be expected between cognitive complexity and the time taken for comprehension if cognitive complexity were a valid measure of understandability.

## 3.2 Correctness

For all data sets, correctness is the percentage of correctly performed tasks from all comprehension tasks.

| DID | Variable | n | Pearson's r | p-value | Spearman's $\rho$ | p-value |
|---|---|---|---|---|---|---|
| 1A | Correctness | 23 | -0.469 | 0.024* | -0.284 | 0.189 |
| 3A | Correctness | 12 | -0.246 | 0.441 | -0.226 | 0.481 |
| 3B | Correctness | 8 | -0.454 | 0.258 | -0.339 | 0.411 |
| 3C | Correctness | 20 | -0.445 | 0.049* | -0.411 | 0.072 |
| 4 | Correctness | 20 | -0.656 | 0.002* | -0.373 | 0.105 |
| 5 | Correctness | 50 | -0.208 | 0.146 | -0.177 | 0.219 |
| 6 | Correctness | 6 | -0.151 | 0.775 | 0.129 | 0.808 |
| 9 | Correctness | 30 | 0.444 | 0.014* | 0.515 | 0.004* |
| 10A | Correctness | 126 | 0.067 | 0.456 | 0.056 | 0.534 |
| 10B | Correctness | 8 | 0.611 | 0.107 | 0.875 | 0.004* |

**Table 3.3:** Raw correlation results for correctness. (*p<.05)

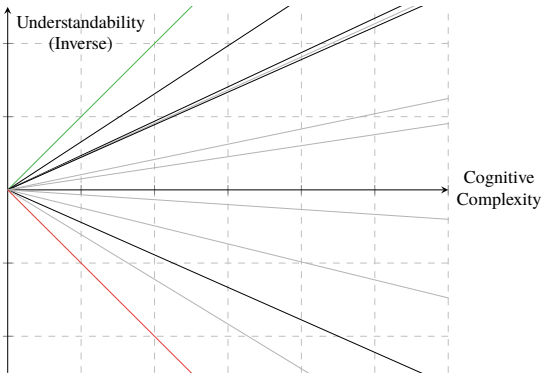


**Figure 3.3:** Interpreted Pearson correlation for correctness

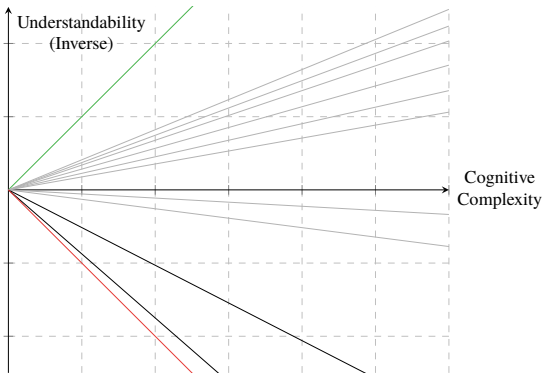


**Figure 3.4:** Interpreted Spearman correlation for correctness

**Black** lines represent significant values (p<0.05) and gray lines represent insignificant values (p>0.05). Again, these lines do not represent the values from Table 3.3 without change but rather the interpreted values with regards to **RQ1**. This means that lines with a positive slope show that a higher cognitive complexity means lower understandability.

Specifically, the correlation values of all correctness variables were inverted since a negative correlation would be expected between cognitive complexity and the correctness of comprehension tasks if cognitive complexity were a valid measure of understandability.

## 3.3 Ratings

All rating variables represent a subjective rating regarding the comprehension task. Pre and Post refer to the time when the subject submitted the rating. Pre meaning before doing the task and post after completing it.

| DID | Variable | n | Pearson's r | p-value | Spearman's $\rho$ | p-value |
|---|---|---|---|---|---|---|
| 1A | Difficulty | 23 | -0.605 | 0.002* | -0.514 | 0.012* |
| 1A | Confidence | 23 | -0.503 | 0.014* | -0.412 | 0.051 |
| 2 | Readability | 100 | -0.283 | 0.004* | -0.202 | 0.044* |
| 5 | Understandability | 50 | -0.065 | 0.654 | -0.018 | 0.903 |
| 6 | Difficulty | 6 | -0.367 | 0.474 | -0.143 | 0.787 |
| 6 | Motivation | 6 | 0.134 | 0.801 | 0.057 | 0.914 |
| 6 | Performance | 6 | -0.068 | 0.898 | -0.071 | 0.893 |
| 9 | Readability (Pre) | 30 | -0.351 | 0.057 | -0.245 | 0.192 |
| 9 | Readability (Post) | 30 | -0.283 | 0.129 | -0.150 | 0.428 |

**Table 3.4:** Raw correlation results for subjective ratings. (*p<.05)

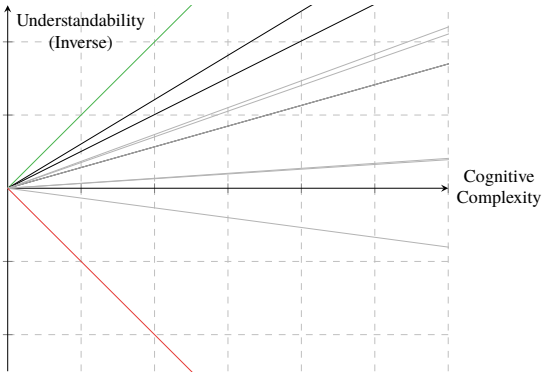


**Figure 3.5:** Interpreted Pearson correlation for ratings

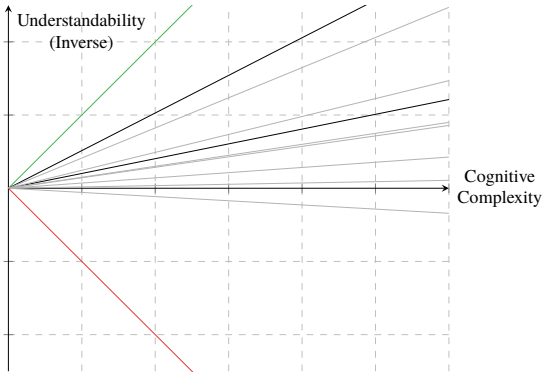


**Figure 3.6:** Interpreted Spearman correlation for ratings

**Black** lines represent significant values (p<0.05) and gray lines represent insignificant values (p>0.05). Again, these lines do not represent the values from Table 3.5 without change but rather the interpreted values with regards to **RQ1**. This means that lines with a positive slope show that a higher cognitive complexity means lower understandability.

Specifically, all correlation values for subjective ratings were inverted. A lower rating meant that the comprehension task was more difficult, the snippet less readable or understandable, or the subject rated their motivation or performance lower. A higher cognitive complexity should result in a lower value for difficulty, readability, and understandability, motivation, and performance. This meant that a negative correlation would be expected if cognitive complexity were a valid measure of understandability.

## 3.4 Physiological Factors

All physiological variables were taken from the study by [PSA+18] and represent the strength of brain deactivation during the comprehension tasks.

| DID | Variable | n | Pearson's r | p-value | Spearman's $\rho$ | p-value |
|---|---|---|---|---|---|---|
| 1B | BA32 | 12 | -0.207 | 0.519 | 0.000 | 1.000 |
| 1B | BA31post | 12 | 0.431 | 0.162 | 0.189 | 0.557 |
| 1B | BA31ant | 12 | -0.020 | 0.952 | -0.098 | 0.762 |

**Table 3.5:** Raw correlation results for physiological measures. (*p<.05)

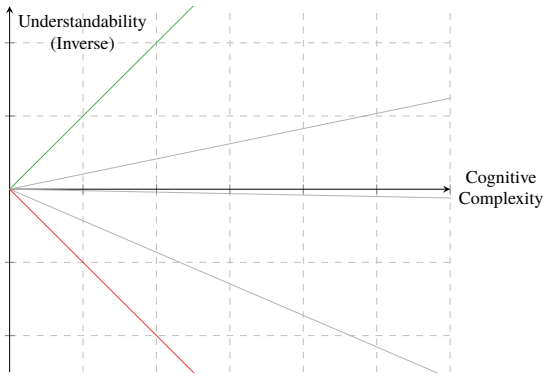


**Figure 3.7:** Interpreted Pearson correlation for physiological factors

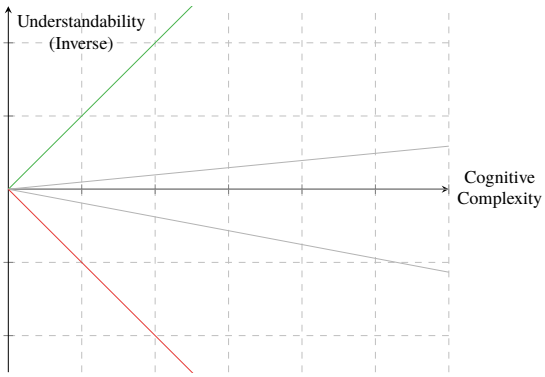


**Figure 3.8:** Interpreted Spearman correlation for physiological factors

**Black** lines represent significant values (p<0.05) and **gray** lines represent insignificant values (p>0.05). Again, these lines do not represent the values from Table 3.5 without change but rather the interpreted values with regards to **RQ1**. This means that lines with a positive slope show that a higher cognitive complexity means lower understandability.

Specifically, all of the correlation values for the physiological values were inverted. This was done because a negative correlation between cognitive complexity and the brain deactivation strength would be expected if cognitive complexity were a valid measure of understandability.

## 3.5 Other Variables

Other variables were made up of composite variables between time and correctness (TAU and TC), binary deceptiveness (BD50) and the number of attempted comprehension tasks (Points).

| DID | Variable | n | Pearson's r | p-value | Spearman's $\rho$ | p-value |
|---|---|---|---|---|---|---|
| 5 | TAU | 50 | -0.321 | 0.023* | -0.309 | 0.029* |
| 5 | BD50 | 50 | 0.236 | 0.099 | 0.378 | 0.007* |
| 10A | TC | 126 | -0.029 | 0.745 | 0.015 | 0.867 |
| 10B | Points | 8 | 0.405 | 0.319 | 0.482 | 0.226 |

**Table 3.6:** Raw correlation results for other variables. (*p<.05)

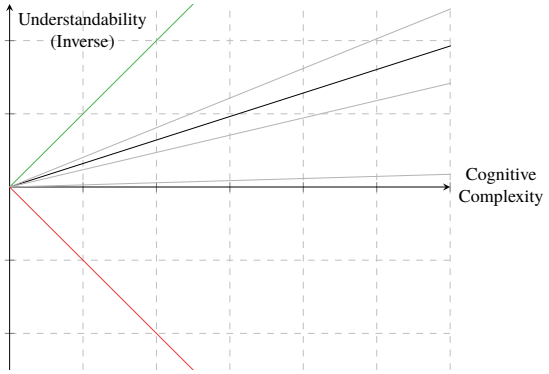


**Figure 3.9:** Interpreted Pearson correlation for other variables

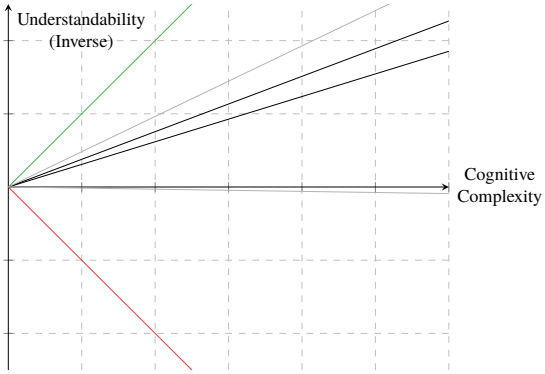


**Figure 3.10:** Interpreted Spearman correlation for other variables

**Black** lines represent significant values (p<0.05) and **gray** lines represent insignificant values (p>0.05). Again, these lines do not represent the values from Table 3.6 without change but rather the interpreted values with regards to **RQ1**. This means that lines with a positive slope show that a higher cognitive complexity means lower understandability.

Specifically, the values for the composite variables TAU and TC were inverted since a lower value would indicate lower understandability.

# 4 Discussion

## 4.1 Systematic Literature Search

As discussed in Chapter 3, most of the studies found in the literature search were published from the year 2011 and onward. The most obvious explanation for this is that the time period for the initial search was set to only include studies from 2010 and later. While this is likely an explanation for the trend observed in this literature search, a study without an initial boundary for the time period might show similar results.

Siegmund [Sie16] describes the same initial rise in popularity of program comprehension studies in the late 1970s and 1980s as well as the eventual decline in the 1990s. They also predict a future surge in research contributions, especially in the realm of empirical experiments due to the recent rise in interest in the field of human factors in software engineering. When comparing the results to other systematic literature reviews in the field of source code understandability, we can find similar results.

Another thing of note for this literature search was the large number of papers that had to be filtered manually. This is the result of the search criteria not being specific enough to automatically exclude the irrelevant papers. Common papers included scientific research from other fields such as comprehension of written text, teaching programs, and line codes. A replication of this literature search might extend the search term to remove common irrelevant terms with the boolean expression `NOT`.

## 4.2 Findings

**RQ1**: *Does cognitive complexity correlate with measures of understandability from existing studies?*

Finding a convincing binary yes or no answer to this question is challenging. For the time variables, the observed results majorly support the hypothesis of **RQ1**. When only considering significant values, for both Spearman's and Pearson's correlation coefficient all values are positive, with 16 out of 20 showing a value of $0.450$ or higher. Their average correlation is $0.654$. While data sets 1B and 7 show a negative correlation, both of them have an extraordinarily high p-value, making their results insignificant. So, to summarize, with regards to the data gathered in this study, cognitive complexity positively and significantly correlates with the time to understand a snippet of code. In other words, the higher the cognitive complexity of source code, the longer it takes to be understood.

The majority of the results for correctness variables also show supporting results. When considering all values, around 13 out of 20 show a negative correlation. As mentioned at the beginning of Chapter 3, a negative correlation for correctness means that the higher the cognitive complexity, the

lower the understandability of the code snippet. It is important to note however, compared to the variables for time, where around 70 % of the results were significant, only 30 % of the correlation results for correctness were significant.

The interpretation of these results is not quite as clear-cut as with the time variables since some of the significant values are opposing the validity of cognitive complexity. The results from data sets 9 and 10B would indicate that an increase in cognitive complexity would improve source code understandability. In their original study, Börstler et al. [BP16] had similar results. They found that with the increase in size of the source code snippets there also was an increase in the correctness of the answers to comprehension questions. If we were to ignore the results from data sets 9 and 10B, the average of the significant correlation values would jump from $0.207$ to $-0.523$. It is difficult to determine why this contrast between the studies exist.

One possible hypothesis is that the results of these studies don't necessarily disprove the validity of cognitive complexity, but rather indicate that some other underlying confounding variables are influencing the correlation between cognitive complexity and understandability. Without further knowledge of these confounding variables, a definitive answer cannot be made about the correlation between cognitive complexity and the correctness of comprehension tasks.

The correlation between cognitive complexity and subject opinions also shows mostly positive results. Again, less than 30 % of the results were significant. Considering only these significant results, they show an average negative correlation of $-0.411$. These include the confidence and difficulty ratings from data set 1A and the perceived readability from data set 2. This means that the higher the cognitive complexity of a code snippet, the lower a subject's confidence in their answer to comprehension questions is and the lower their rating for readability and understandability of the snippet becomes. When also considering the results of the insignificant correlation factors, all of them also show supporting results, except motivation. Here, a higher value in cognitive complexity correlated with a rating describing a higher amount of motivation.

For the physiological variables, no significant correlation could be found between cognitive complexity and concentration during comprehension tasks. This is in line with the results from the original study conducted by Peitek et al. [PSA+18] where the data came from. Cognitive complexity performs similarly to cyclomatic complexity, but worse than DepDegree and Halstead's measures. It is important to keep in mind however, that the sample size of the original study was fairly small and such the interpretation of the data is to be done with caution. Peitek et al. also mentioned this in their initial study. It would be interesting to see how cognitive complexity performs in larger studies and with other physiological measures such as EDA, EEG or others.

In summary, most of the results gathered in this study seem to support the validity of cognitive complexity as a measure of source code understandability. There is still room for improvement, however, since most results from this study still only show moderately strong correlations around $0.4$ to $0.7$. More studies with a larger focus on the characteristics of the source code snippets might show insight into how cognitive complexity can be improved even further. This is of special importance in light of the statistics shown in Table 3.1. The range of cognitive complexity varied wildly between the snippets of existing studies. While some show a large range of values from zero to around forty to fifty, others don't exceed a value of even three or four.

Additionally, combining metrics shows promising results in general [TCM+18] and including cognitive complexity in a combined model might bring us close to an even more accurate measure for source code understandability.

## 4.3 Threats to Validity

One possible threat to validity is the fact that this study solely relies on the data gathered from existing experiments. While the selected data sets were filtered to only include results from peer-reviewed studies, there was no additional quality filtering during the literature search. This means that the methodology of each of the studies was not explicitly examined to find errors. Any incorrect measurements from methodological faults will carry over to the results of this study.

An important aspect to note when considering the results of this analysis is the heterogeneity of the studies that formed the basis for the analysis. Researchers use different terminology, measures, and evaluations making it difficult to find a common reference point for comparison. This notion is supported by the literature review conducted by Schröter et al. [SKSL17]. They stress that the terminology used to report studies is often ambiguous, making it difficult to compare them. The result of this is that some deviations in methodology might influence the results gathered in this work. Considering this shortcoming, future endeavors in studying source code understandability should adhere to a common methodological framework to make them more comparable.

For systematic literature reviews, often multiple researchers go through the search process and measure their agreement [KC07]. For this study, the systematic literature search was conducted by a single researcher. The effect of this shortcoming is that any human errors, especially in the manual filtering process, could not be fixed by peer-review. To mitigate this, the search strategy and methods were documented thoroughly before the beginning of the search to make sure that there is as little deviation from the planned steps as possible.

# 5 Conclusion

The goal of this work was to find evidence of whether or not cognitive complexity correlates with measures of understandability from existing studies. To find a comprehensive basis of data, a systematic literature search was conducted to find studies that measure the understandability of source code snippets from a human point of view. The result was 14 studies spanning over 324 code snippets and approximately 24,400 individual human evaluations which could be used for data analysis. The data points were used to calculate the correlation between the cognitive complexity of the code snippets and the corresponding measures of understandability. Results from this analysis show that overall, cognitive complexity correlates significantly with most measures of understandability from existing studies.

This shows promising prospects for future research on cognitive complexity. One factor of interest could be how well cognitive complexity performs in a real-world case study. While the original paper by Campbell et al. [Cam17] mentioned an initial study of developer acceptance of the metric, a focused field study on the evolution of code understandability when employing methods to reduce cognitive complexity could provide further insights on the practical appliance of understandability-focused metrics. Specifically, one would measure the understandability and cognitive complexity of code over the course of a real project and instruct developers to change code to remove high cognitive complexity when discovered (e.g. using SonarCloud). The hypotheses to investigate here would be whether understandability of the code base of a real-world project would improve when employing focused methods to reduce cognitive complexity.

In the future, a study with a focus on code snippets could also be of relevance to the metric. One could look at which code features influence cognitive complexity in order to further improve the correlation between cognitive complexity and understandability. It could also be useful to find a threshold where the cognitive complexity of a snippet is too high and the code should be improved to keep high understandability. Additionally, the use of physiological measures in understandability studies could also provide more insight into how we can measure understandability from the human point of view. Data from studies with larger sample sizes might prove useful in future attempts to correlate source code metrics with these measures. One more possibility is the use of other metrics in combination with cognitive complexity to further improve the correlation with understandability measures. Trockman et al. [TCM+18] showed that in theory, an effort to employ combined metric models might bring us even closer to an accurate source code-based measure of understandability.

All in all, the knowledge gained from this work shows promise for the future of understandability measures. As far as we know, cognitive complexity is the first solely code-based metric that correlates with understandability in a meaningful way. Still, with this in mind, cognitive complexity is a comparatively small piece of the larger puzzle that is the paradigm of code comprehension. But with new ways to measure understanding on a physiological level and promising attempts to combine metrics, creating an accurate measure of code understandability seems close within our grasp.

# Bibliography


[AKGA11] M. Abbes, F. Khomh, Y. Gueheneuc, G. Antoniol. “An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, on Program Comprehension”. In: *2011 15th European Conference on Software Maintenance and Reengineering*. Mar. 2011, pp. 181–190. DOI: 10.1109/CSMR.2011.24 (cit. on p. 37).

[AMS12] S. Aljunid, A. Mohd. Zin, Z. Shukur. English. In: *Journal of Software Engineering* 6.1 (2012), pp. 1–9. ISSN: 1819-4311. DOI: 10.3923/jse.2012.1.9 (cit. on p. 17).

[AWF17] S. Ajami, Y. Woodbridge, D. G. Feitelson. “Syntax, Predicates, Idioms - What Really Affects Code Complexity?” In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. May 2017, pp. 66–76. DOI: 10.1109/ICPC.2017.39 (cit. on pp. 37, 38).

[BBL76] B. W. Boehm, J. R. Brown, M. Lipow. “Quantitative Evaluation of Software Quality”. In: *Proceedings of the 2Nd International Conference on Software Engineering*. ICSE ’76. San Francisco, California, USA: IEEE Computer Society Press, 1976, pp. 592–605 (cit. on pp. 15, 16).

[BP16] J. Börstler, B. Paech. “The Role of Method Chains and Comments in Software Readability and Comprehension—An Experiment”. In: *IEEE Transactions on Software Engineering* 42.9 (Sept. 2016), pp. 886–898. ISSN: 0098-5589. DOI: 10.1109/TSE.2016.2527791 (cit. on pp. 37, 38, 52).

[BW08] R. P. Buse, W. R. Weimer. “A Metric for Software Readability”. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ISSTA ’08. Seattle, WA, USA: ACM, 2008, pp. 121–130. ISBN: 978-1-60558-050-0. DOI: 10.1145/1390630.1390647 (cit. on pp. 37, 38).

[BW10] R. P. L. Buse, W. R. Weimer. “Learning a Metric for Code Readability”. In: *IEEE Transactions on Software Engineering* 36.4 (July 2010), pp. 546–558. ISSN: 0098-5589. DOI: 10.1109/TSE.2009.70 (cit. on pp. 17, 37, 38).

[Cam17] G. A. Campbell. “Cognitive Complexity-A new way of measuring understandability”. In: *Technical Report. SonarSource SA, Switzerland* (2017) (cit. on pp. 20, 55).

[Cam18] G. A. Campbell. “Cognitive Complexity: An Overview and Evaluation”. In: *Proceedings of the 2018 International Conference on Technical Debt*. TechDebt ’18. Gothenburg, Sweden: ACM, 2018, pp. 57–58. ISBN: 978-1-4503-5713-5. DOI: 10.1145/3194164.3194186 (cit. on pp. 15, 20, 21).

[Cor89] T. A. Corbi. “Program understanding: Challenge for the 1990s”. In: *IBM Systems Journal* 28.2 (1989), pp. 294–306. DOI: 10.1147/sj.282.0294 (cit. on p. 15).

[DHOH03] J. J. Dolado, M. Harman, M. C. Otero, L. Hu. "An empirical investigation of the influence of a type of side effects on program comprehension". In: *IEEE Transactions on Software Engineering* 29.7 (July 2003), pp. 665–670. ISSN: 0098-5589. DOI: 10.1109/TSE.2003.1214329 (cit. on pp. 37, 38).

[FALK11] J. Feigenspan, S. Apel, J. Liebig, C. Kastner. "Exploring Software Measures to Assess Program Comprehension". In: *2011 International Symposium on Empirical Software Engineering and Measurement*. Sept. 2011, pp. 127–136. DOI: 10.1109/ESEM.2011.21 (cit. on pp. 15, 17, 19, 37, 38).

[FBM+14] T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, M. Züger. "Using Psycho-physiological Measures to Assess Task Difficulty in Software Development". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 402–413. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568266 (cit. on pp. 18, 37, 38).

[FGN+19] D. Fucci, D. Girardi, N. Novielli, L. Quaranta, F. Lanubile. "A Replication Study on Code Comprehension and Expertise Using Lightweight Biometric Sensors". In: *Proceedings of the 27th International Conference on Program Comprehension*. ICPC '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 311–322. DOI: 10.1109/ICPC.2019.00050 (cit. on pp. 18, 37, 38).

[FMAA18] S. Fakhoury, Y. Ma, V. Arnaoudova, O. Adesope. "The Effect of Poor Source Code Lexicon and Readability on Developers' Cognitive Load". In: *Proceedings of the 26th Conference on Program Comprehension*. ICPC '18. Gothenburg, Sweden: ACM, 2018, pp. 286–296. ISBN: 978-1-4503-5714-2. DOI: 10.1145/3196321.3196347 (cit. on pp. 37, 38).

[FSW17] B. Floyd, T. Santander, W. Weimer. "Decoding the Representation of Code in the Brain: An fMRI Study of Code Review and Expertise". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. May 2017, pp. 175–186. DOI: 10.1109/ICSE.2017.24 (cit. on pp. 17, 37, 38).

[GIY+17] D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K.-C. Yeh, J. Cappos. "Understanding Misunderstandings in Source Code". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 129–139. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3106264 (cit. on pp. 37, 38).

[HS88] S. G. Hart, L. E. Staveland. "Development of NASA-TLX (Task Load Index): Results of Empirical and Theoretical Research". In: *Human Mental Workload*. Ed. by P. A. Hancock, N. Meshkati. Vol. 52. Advances in Psychology. North-Holland, 1988, pp. 139–183. DOI: 10.1016/S0166-4115(08)62386-9 (cit. on p. 34).

[HSH17] J. Hofmeister, J. Siegmund, D. V. Holt. "Shorter identifier names take longer to comprehend". In: *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. Feb. 2017, pp. 217–227. DOI: 10.1109/SANER.2017.7884623 (cit. on pp. 17, 37, 38).

[IEC01] I. IEC. "9126-1 (2001). Software Engineering Product Quality-Part 1: Quality Model". In: *International Organization for Standardization* (2001) (cit. on p. 16).

[Iso11] I. Iso. “ISO-IEC 25010: 2011 Systems and Software Engineering-Systems and Software Quality Requirements and Evaluation (SQuaRE)-System and Software Quality Models”. In: *International Organization for Standardization* 34 (2011), p. 2910 (cit. on p. 16).

[KC07] B. Kitchenham, S. Charters. *Guidelines for performing systematic literature reviews in software engineering*. Tech. rep. Technical report, Ver. 2.3 EBSE Technical Report. EBSE, 2007 (cit. on pp. 23, 27, 53).

[KW13] N. Kasto, J. Whalley. “Measuring the Difficulty of Code Comprehension Tasks Using Software Metrics”. In: *Proceedings of the Fifteenth Australasian Computing Education Conference - Volume 136*. ACE ’13. Adelaide, Australia: Australian Computer Society, Inc., 2013, pp. 59–65. ISBN: 978-1-921770-21-0 (cit. on pp. 17, 18).

[Lik32] R. Likert. “A technique for the measurement of attitudes.” In: *Archives of Psychology* 22 140 (1932), pp. 55–55 (cit. on p. 34).

[McC76] T. J. McCabe. “A Complexity Measure”. In: *IEEE Transactions on Software Engineering* SE-2.4 (Dec. 1976), pp. 308–320. DOI: 10.1109/TSE.1976.233837 (cit. on p. 19).

[MML15] R. Minelli, A. Mocci, M. Lanza. “I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time”. In: *2015 IEEE 23rd International Conference on Program Comprehension*. May 2015, pp. 25–35. DOI: 10.1109/ICPC.2015.12 (cit. on p. 15).

[PSA+18] N. Peitek, J. Siegmund, S. Apel, C. Kästner, C. Parnin, A. Bethmann, T. Leich, G. Saake, A. Brechmann. “A Look into Programmers’ Heads”. In: *IEEE Transactions on Software Engineering* (2018), pp. 1–1. ISSN: 0098-5589. DOI: 10.1109/TSE.2018.2863303 (cit. on pp. 18, 37, 38, 48, 52).

[RCM15] L. B. A. Rabai, B. Cohen, A. Mili. “Programming language use in us academia and industry”. In: *Informatics in Education* 14.2 (2015), p. 143. DOI: 10.15388/infedu.2015.09 (cit. on p. 36).

[RTYB85] C. Ramamoorthy, W. T. Tsai, T. Yamaura, A. Bhide. “Metrics Guided Methodology”. In: *Proceedings-IEEE Computer Society’s International Computer Software & Applications Conference*. IEEE. 1985, pp. 111–120 (cit. on p. 17).

[RW97] V. Ramalingam, S. Wiedenbeck. “An Empirical Study of Novice Program Comprehension in the Imperative and Object-oriented Styles”. In: *Papers Presented at the Seventh Workshop on Empirical Studies of Programmers*. ESP ’97. Alexandria, Virginia, USA: ACM, 1997, pp. 124–139. ISBN: 0-89791-992-0. DOI: 10.1145/266399.266411 (cit. on p. 17).

[SAPM14] G. Salvaneschi, S. Amann, S. Proksch, M. Mezini. “An Empirical Study on Program Comprehension with Reactive Programming”. In: *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: ACM, 2014, pp. 564–575. ISBN: 978-1-4503-3056-5. DOI: 10.1145/2635868.2635895 (cit. on pp. 37, 38).

[SBV+17] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, R. Oliveto. "Automatically assessing code understandability: How far are we?" In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Oct. 2017, pp. 417–427. DOI: 10.1109/ASE.2017.8115654 (cit. on pp. 19, 37, 38).

[SBV+19] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-V?squez, D. Poshyvanyk, R. Oliveto. "Automatically Assessing Code Understandability". In: *IEEE Transactions on Software Engineering* (2019), pp. 1–1. ISSN: 0098-5589. DOI: 10.1109/TSE.2019.2901468 (cit. on pp. 15, 17, 19, 37, 38).

[Sed16] T. Sedano. "Code Readability Testing, an Empirical Study". In: *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*. Apr. 2016, pp. 111–117. DOI: 10.1109/CSEET.2016.36 (cit. on p. 17).

[She88] M. Shepperd. "A critique of cyclomatic complexity as a software metric". In: *Software Engineering Journal* 3.2 (Mar. 1988), pp. 30–36. DOI: 10.1049/sej.1988.0003 (cit. on p. 20).

[Sie16] J. Siegmund. "Program Comprehension: Past, Present, and Future". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 5. Mar. 2016, pp. 13–20. DOI: 10.1109/SANER.2016.35 (cit. on pp. 17, 51).

[SKA+14] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, A. Brechmann. "Understanding Understanding Source Code with Functional Magnetic Resonance Imaging". In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: ACM, 2014, pp. 378–389. ISBN: 978-1-4503-2756-5. DOI: 10.1145/2568225.2568252 (cit. on pp. 17, 37, 38).

[SKSL17] I. Schröter, J. Krüger, J. Siegmund, T. Leich. "Comprehending Studies on Program Comprehension". In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. May 2017, pp. 308–311. DOI: 10.1109/ICPC.2017.9 (cit. on p. 53).

[SSA13] M. M. Suleman Sarwar, S. Shahzad, I. Ahmad. "Cyclomatic complexity: The nesting problem". In: *Eighth International Conference on Digital Information Management (ICDIM 2013)*. Sept. 2013, pp. 274–279. DOI: 10.1109/ICDIM.2013.6693981 (cit. on p. 20).

[Tay53] W. L. Taylor. "Cloze procedure: a new tool for measuring readability." In: *Journalism Bulletin* 30.4 (1953), pp. 415–433. DOI: 10.1177/107769905303000401 (cit. on p. 34).

[TCM+18] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner, B. Vasilescu. "Automatically Assessing Code Understandability Reanalyzed: Combined Metrics Matter". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: ACM, 2018, pp. 314–318. ISBN: 978-1-4503-5716-6. DOI: 10.1145/3196398.3196441 (cit. on pp. 19, 52, 55).

[TFSL14] R. Turner, M. Falcone, B. Sharif, A. Lazar. "An Eye-tracking Study Assessing the Comprehension of C++ and Python Source Code". In: *Proceedings of the Symposium on Eye Tracking Research and Applications*. ETRA '14. Safety Harbor, Florida: ACM, 2014, pp. 231–234. ISBN: 978-1-4503-2751-0. DOI: 10.1145/2578153.2578218 (cit. on pp. 18, 37, 38).

[WDS81] S. N. Woodfield, H. E. Dunsmore, V. Y. Shen. “The Effect of Modularization and Comments on Program Comprehension”. In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE ’81. San Diego, California, USA: IEEE Press, 1981, pp. 215–223. ISBN: 0-89791-146-6 (cit. on p. 17).

[Wey88] E. J. Weyuker. “Evaluating software complexity measures”. In: *IEEE Transactions on Software Engineering* 14.9 (Sept. 1988), pp. 1357–1365. DOI: 10.1109/32.6178 (cit. on p. 20).

[Woh14] C. Wohlin. “Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering”. In: *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. EASE ’14. London, England, United Kingdom: ACM, 2014, 38:1–38:10. ISBN: 978-1-4503-2476-2. DOI: 10.1145/2601248.2601268 (cit. on p. 27).

[WR99] S. Wiedenbeck, V. Ramalingam. “Novice comprehension of small programs written in the procedural and object-oriented styles”. In: *International Journal of Human-Computer Studies* 51.1 (1999), pp. 71–87. ISSN: 1071-5819. DOI: 10.1006/ijhc.1999.0269 (cit. on p. 17).

[WRSC99] S. Wiedenbeck, V. Ramalingam, S. Sarasamma, C. Corritore. “A comparison of the comprehension of object-oriented and procedural programs by novice programmers”. In: *Interacting with Computers* 11.3 (1999), pp. 255–282. ISSN: 0953-5438. DOI: 10.1016/S0953-5438(98)00029-0 (cit. on p. 17).

[XBL+18] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, S. Li. “Measuring Program Comprehension: A Large-Scale Field Study with Professionals”. In: *IEEE Transactions on Software Engineering* 44.10 (Oct. 2018), pp. 951–976. DOI: 10.1109/TSE.2017.2734091 (cit. on p. 15).

[YDM+05] Yaofei Chen, R. Dios, A. Mili, Lan Wu, Kefei Wang. “An empirical study of programming language trends”. In: *IEEE Software* 22.3 (May 2005), pp. 72–79. DOI: 10.1109/MS.2005.55 (cit. on p. 36).

[YGYZ17] M. K. .-.-. Yeh, D. Gopstein, Y. Yan, Y. Zhuang. “Detecting and comparing brain activity in short program comprehension using EEG”. In: *2017 IEEE Frontiers in Education Conference (FIE)*. Oct. 2017, pp. 1–5. DOI: 10.1109/FIE.2017.8190486 (cit. on p. 18).

All links were last followed on October 24, 2019.

**Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature