The Journal of Systems & Software 197 (2023) 111561

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

An empirical evaluation of the "Cognitive Complexity" measure as a predictor of code understandability☆

Luigi Lavazza [a,*], Abedallah Zaid Abualkishik [b], Geng Liu [c], Sandro Morasca [a]

[a] Dipartimento di Scienze Teoriche e Applicate, Università degli Studi dell'Insubria, Varese, Italy
[b] College of Computer Information Technology, American University in the Emirates, Dubai, United Arab Emirates
[c] School of Computer Science and Technology, Hangzhou Dianzi University, China

ARTICLE INFO

Article history:
Received 2 October 2021
Received in revised form 7 September 2022
Accepted 16 November 2022
Available online 19 November 2022

Keywords:
Software understandability
Cognitive complexity
Software code measures
Complexity measures
Static code measures

ABSTRACT

Background: Code that is difficult to understand is also difficult to inspect and maintain and ultimately causes increased costs. Therefore, it would be greatly beneficial to have source code measures that are related to code understandability. Many "traditional" source code measures, including for instance Lines of Code and McCabe's Cyclomatic Complexity, have been used to identify hard-to-understand code. In addition, the "Cognitive Complexity" measure was introduced in 2018 with the specific goal of improving the ability to evaluate code understandability.
Aims: The goals of this paper are to assess whether (1) "Cognitive Complexity" is better correlated with code understandability than traditional measures, and (2) the availability of the "Cognitive Complexity" measure improves the performance (i.e., the accuracy) of code understandability prediction models.
Method: We carried out an empirical study, in which we reused code understandability measures used in several previous studies. We first built Support Vector Regression models of understandability vs. code measures, and we then compared the performance of models that use "Cognitive Complexity" against the performance of models that do not.
Results: "Cognitive Complexity" appears to be correlated to code understandability approximately as much as traditional measures, and the performance of models that use "Cognitive Complexity" is extremely close to the performance of models that use only traditional measures.
Conclusions: The "Cognitive Complexity" measure does not appear to fulfill the promise of being a significant improvement over previously proposed measures, as far as code understandability prediction is concerned.

1. Introduction

Program comprehension absorbs a large part of the time and effort spent by professionals during software development (Minelli et al., 2015; Xia et al., 2017). Thus, knowing whether there is a relationship between code understandability and some code measures would be greatly beneficial for software development: software modules that appear hard to understand could be revised, to improve their readability, thus favoring subsequent maintenance activities. In addition, coding rules could be established based on code measures, to proactively keep understandability adequate and make maintenance less painful.

Code measures such as McCabe's Complexity (also known as cyclomatic complexity, which we denote as *McCC*) (McCabe, 1976), various Maintainability Indices (Oman and Hagemeister, 1992; Heitlager et al., 2007), and Halstead measures (Halstead, 1977) have been used to this end in the past. However, none of them has turned out to be effective enough to predict and control code understandability. A new source code measure, called "Cognitive Complexity", was introduced in 2018 (Campbell, 2018a) with the goal of assessing code understandability, while overcoming some of the pitfalls of existing code measures, notably *McCC*. In the remainder of this paper, we denote this measure as *CoCo*.

However, the proposers of *CoCo* did not provide solid empirical evidence that *CoCo* is actually better than existing code measures in indicating code that is difficult to understand. Our goal here is to seek for evidence that supports or contradicts the claims that accompanied the proposal of *CoCo*, and specifically the fact that it is more effective in predicting understandability than existing measures. This is common practice (Shen et al., 1983; Shepperd, 1988; Subramanyam and Krishnan, 2003; Denaro et al., 2003; Padilha et al., 2014), as it is not simply sufficient to show that the newly introduced measure *CoCo* is itself related to understandability. This certainly is a necessary, but preliminary step, which

☆ Editor: Matthias Galster.

* Corresponding author.
E-mail addresses: luigi.lavazza@uninsubria.it (L. Lavazza), abedallah.abualkishik@aue.ae (A.Z. Abualkishik), liugeng@hdu.edu.cn (G. Liu), sandro.morasca@uninsubria.it (S. Morasca).

https://doi.org/10.1016/j.jss.2022.111561
0164-1212/

must be followed by providing evidence that *CoCo* is also better than the existing measures.

In this paper, we report on an empirical study that we carried out to investigate whether *CoCo* provides a real improvement in the assessment of code understandability. Before adopting any measure for this purpose, practitioners need to know how effectively it can help them spot hard-to-understand software code. First, there are costs incurred for the collection of the measure, related to acquiring new tools, updating historical datasets, updating data analysis procedures, etc. So, practitioners need to know if adopting the measure is worth the effort. Second, and perhaps more important, adopting a new measure for practical use entails taking actions based on it. For instance, one can set a threshold on the measure to separate supposedly clear code from hard-to-understand code, so that the latter may undergo modifications. All of this is justified and beneficial only if there is solid evidence that that measure is truly related to understandability. Lacking this evidence, using the measure during software development may increase software cost and, possibly, even lower software quality as the result of needless modifications.

Therefore, our empirical study first addresses the following Research Question, thus extending the work by Barón et al. (2020a) by considering "traditional" measures:

RQ1 How well is *CoCo* correlated with software understandability, also in comparison with "traditional" measures?

Since it would be impossible to answer RQ1 for the thousands of source code measures that have been defined in the past, we concentrate on the source code measures provided by a state-of-the-art measurement tool, which measures code properties similar to those considered by *CoCo*, or have been proposed and widely used as predictors of code understandability. Thus, these are measures that are likely to be used in a software production environment. An empirical study showing that *CoCo* provides substantial improvements over existing measures would give *CoCo* greater credentials to be applied in practice. If, instead, this were not the case, *CoCo* would appear to be much less useful, since other existing measures perform as well as it does, if not better.

CoCo was introduced as an alternative to existing source code measures, though without a real empirical or theoretical validation. A first evaluation of how well *CoCo* can be used to assess code understandability was performed by Barón et al. (2020a). They collected published data from empirical studies on code understandability, measured the *CoCo* of the source code used in the experiments, and evaluated the statistical association between various types of understandability indicators and *CoCo* (see Section 3.2). They found moderate associations in some cases.

In this paper, we investigate the claim based on which *CoCo* was introduced, i.e., that it is better suited than existing source code measures for assessing code understandability, whereas the goal of Muñoz Barón et al.'s article (Barón et al., 2020a) was to assess *CoCo per se*.

Thus, we addressed the following second Research Question in our empirical study.

RQ2 Using *CoCo*, is it possible to build better predictors of code understandability than using only traditional source code measures?

To answer both Research Questions, we reuse the data collected by Muñoz Barón et al. from several papers reporting the results of experiments involving the evaluation of different aspects of code understandability.

The remainder of this paper is organized as follows. Section 2 provides some background on code understandability and its measurement. Section 3 reviews the source code measures we use in our empirical study and specifically *CoCo*, which is the most recent one, along with some of its precursors. The empirical study is described in Section 4 and its results are illustrated in Section 5. In Section 6 we answer the Research Questions. The threats to the validity of the empirical study are discussed in Section 7. Section 8 illustrates the conclusions and outlines future work. Appendices A and B provide further details about the building of prediction models and the evaluation of our results.

2. Source code understandability

Software maintenance is responsible for the majority of software costs (60% on average, according to Glass, 2001). The unfamiliarity of maintainers with the software code they need to maintain is one of the main reasons why they must spend a disproportionate part of their time understanding code (Minelli et al., 2015). So, it is hardly surprising that software maintainability and understandability have been widely studied and measures for them have been proposed.

Qualities like maintainability or understandability are so-called "external" software attributes (Fenton and Bieman, 2014), since they depend on the knowledge of both the software code at hand and its relationships with its "environment", i.e., how and by whom it is maintained or understood. For instance, maintaining or understanding a piece of software code is generally less effort-consuming for the developer that originally wrote it than for other developers.

In this paper, we focus on understandability, whose measures play the role of dependent variables in our empirical study. These measures may quantify different aspects of understandability. For instance, a measure may be related to the effort needed to understand some software code, while another may address the depth with which that software code was understood. Thus, software understandability has been quantified in a variety of ways in the literature.

Here, we summarize the ways in which understandability is measured in the primary studies that we use in our empirical analysis, i.e., the understandability aspects that were taken into account. We review these studies in more detail in Section 4.1. Each of these ways corresponds to a different aspect of understandability.

- *Time*. A first typical understandability measure is the time taken to carry out some comprehension task on some software code. This measure was used in all of the primary studies we considered except (Buse and Weimer, 2010; Gopstein et al., 2017).
- *Correctness*. Correctness is usually defined operationally. Specifically, some tasks that require understanding a specific portion of software code are defined first, and the degree of success in performing these tasks is taken as an understandability measure. This practice is widely adopted (Siegmund et al., 2012; Dolado et al., 2003; Salvaneschi et al., 2014; Scalabrino et al., 2021; Börstler and Paech, 2016). However, different authors carried out different experiments involving different tasks, hence there is no unique definition of correctness. As an example, correctness can be assessed by (1) having a maintainer read a piece of code, (2) asking the maintainer a number of questions on the code, and (3) counting the number of correct answers.
- Subjective *rating*. Maintainers can be asked to rate their subjective perception of how well they understood the given code, typically on an ordinal scale. This kind of ratings was used in Buse and Weimer (2010), Scalabrino et al. (2021) and Börstler and Paech (2016).

– *Physiological* measures. A number of studies (Ikutani and Uwano, 2014; Floyd et al., 2017; Fucci et al., 2019; Sharafi et al., 2021) investigated the physiological activities occurring in the human body when understanding software code, involving for instance the brain, heart, and skin. Physiological measures were also used to quantify understandability in one of the primary studies whose data we reuse (Peitek et al., 2020).

The measures used in our empirical study for these four aspects of understandability are described in Section 4.1. The details about the definitions of these measures can be found in the aforementioned papers.

3. Source code structural measures

Many software measures have been defined to capture so-called "internal" software attributes (Fenton and Bieman, 2014), which are defined as those attributes of an entity (source code, in our case) that can be measured based only on the knowledge of the entity. Examples of internal software attributes are size, complexity, cohesion, and coupling. The measures of internal attributes are especially useful when they are associated with some process variable of interest (e.g., software development cost) or with some external software attribute (Fenton and Bieman, 2014; Morasca, 2009) (e.g., software understandability).

In our empirical study, we investigate whether a number of source code measures are correlated with the understandability measures described in Section 2. Specifically, we considered the following measures.

Logical Lines Of Code The lines of code are the first characteristic of code that was measured. *LOC* (or the logical *LOC*, i.e., *LLOC*) are so widely used that performing a study on code measures without considering *LOC* (or *LLOC*) is almost inconceivable.

McCabe's Cyclomatic Complexity *McCC* was originally proposed to identify software modules that are difficult to test or maintain, by counting the independent paths in the control flow graph of code. It has been used extensively as an indicator of difficult understandability and maintainability. In the introduction of the document that proposed "Cognitive Complexity" (Campbell, 2018a), *McCC* is depicted as a measure that is inadequate to represent code understandability, while "Cognitive Complexity" addresses the inadequacies of McCabe's complexity. It is therefore interesting to evaluate to what extent (if at all) *CoCo* achieves the purpose of successfully identifying the code that is hard to understand.

Nesting Level Else-If Nesting Level Else-If (*NLE*) measures the depth of the maximum embeddedness of a method's conditional, iteration, and exception handling block scopes, whereas in the if-else-if construct only the first if instruction is considered. Deep nesting of control structures, hence a high value of *NLE*, is expected to make code harder to understand.

As we show in Section 3.1, *CoCo* accounts for both the decision points (which are the basic elements for measuring *McCC*) and the nesting level (which is measured by *NLE*). Hence, it is reasonable to expect that *CoCo* is able to identify unreadable code more effectively than each of *McCC* and *NLE*. However, it is not clear whether *McCC* and *NLE* together may achieve better results, or if *McCC*, *NLE*, and *CoCo* together may be even more accurate at discovering unreadable code. Our study provides some evidence.

Table 1
The measures from SourceMeter that we used.

Measure name	Abbreviation
Halstead Calculated Program Length	HCPL
Halstead Volume	HVOL
Maintainability Index	MI
McCabe's Cyclomatic Complexity	McCC
Nesting Level Else-If	NLE
Logical Lines of Code	LLOC

Several software measures were proposed long ago and have been widely used (with varying levels of success) to identify hard-to-understand code. It is thus interesting to verify also whether *CoCo* provides better performance compared to those metrics. We considered the following ones.

HVOL Halstead identified measurable properties of software in analogy with the measurable properties of matter (Halstead, 1977). Halstead Volume (*HVOL*) is computed as follows:

$$HVOL = N * \log_2(\eta) \quad (1)$$

where $N = N_1 + N_2$ is the "program length", N_1 is the total number of operators, N_2 is total number of operands; $\eta = \eta_1 + \eta_2$ is the "program vocabulary", η_1 is the number of distinct operators and η_2 is the number of distinct operands. According to Fitzsimmons and Love (1978), "For each of the N elements of a program, $\log_2\ \eta$ bits must be specified to choose one of the operators or operands for that element. Thus *HVOL* measures the number of bits required to specify a program."

HCPL Halstead Calculated Program Length (*HCPL*) is computed as follows:

$$HCPL = \eta_1 * \log_2(\eta_1) + \eta_2 * \log_2(\eta_2) \quad (2)$$

Maintainability Index The Maintainability Index, whose original definition by Coleman et al. (1994) was then simplified by Welker et al. (1997), is computed by the following formula:

$$MI = 171 - 5.2 * \ln(HVOL) - 0.23 * (McCC) - 16.2 * \ln(LLOC) \quad (3)$$

The usage of the measures described above for maintainability evaluation was evaluated by several authors, and some of these measures were considered inappropriate (see for instance the discussion by Ostberg and Wagner, 2014). Nonetheless, we included these measures in our empirical study as a sort of benchmark for the evaluation of *CoCo*.

In summary, we used *CoCo*, which is described in more detail in Section 3.1, and the measures listed in Table 1, which are collected by SourceMeter.[1]

We used SourceMeter because it is a fairly consolidated and robust tool, it supports all of the programming languages involved in our analysis (except Scala), and it is efficient and well documented.

3.1. The "Cognitive Complexity" measure

In 2018, SonarSource introduced "Cognitive Complexity" (Campbell, 2018a) as a new measure for the understandability of a given piece of code. *CoCo* takes into account several aspects

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

of code. Like McCabe's complexity, it takes into account decision points (conditional statements, loops, switch statements, etc.), but, unlike McCabe's complexity, gives them a weight equal to their nesting level plus 1. So, for instance, in the following code fragment

```
void firstMethod() {
 if (condition1)
  for (int i = 0; i < 10; i++)
   while (condition2) { ... }
}
```

the `if` statement at nesting level 0 has weight 1, the `for` statement at nesting level 1 has weight 2, and the `while` statement at nesting level 2 has weight 3, thus $CoCo = 1 + 2 + 3 = 6$. The same code has $McCC = 4$ (3 decision points+1).

Consider instead the following code fragment, in which the three control structures are not nested.

```
void secondMethod() {
 if (condition1) { ... }
 for (int i = 0; i < 10; i++) { ... }
 while (condition2) { ... }
}
```

It has $CoCo = 3$, because all the three control instructions are at nesting level 0 and have weight 1; its McCabe complexity is still $McCC = 4$. It is thus apparent that nested structures increase *CoCo*, while they have no effect on *McCC*.

Boolean predicates are also taken into account. Specifically, a Boolean predicate contributes to *CoCo* depending on the number of its sub-sequences of logical operators. For instance, consider the following code fragment, where `a, b, c, d, e, f` are Boolean variables:

```
void thirdMethod() {
 if (a && b && c || d || e && f) { ... }
}
```

Predicate `a && b && c || d || e && f` contains three subsequences with the same logical operators, i.e., `a && b && c`, `... || d || e`, and `... && f`, so it adds 3 to the value of *CoCo*.

Other aspects of code contribute to incrementing *CoCo*, but they are much less frequent than those described above. For a complete description of *CoCo*, see the definition (Campbell, 2018a).

As a sort of validation of the proposed measure, Campbell performed an investigation of the developers' reaction to the introduction of *CoCo* in the measurement and analysis tool SonarCloud. In an analysis of 22 open-source projects, they assessed whether a development team "accepted" the measure, based on whether they fixed those code areas indicated by the tool as characterized by high *CoCo*. Around 77% of developers expressed acceptance of the measure (Campbell, 2018b).

The next section describes the first study that provided a validation of *CoCo*.

3.2. Empirical validation of CoCo

An objective validation of the *CoCo* measure was performed by Barón et al. (2020a). In the first phase of their work, they conducted a literature search to find datasets from published studies that measured the understandability of source code from the perspective of human developers. They selected the papers with publicly available datasets, and collected the data concerning various aspects of understandability, as well as the code snippets used in the experiments carried in the selected papers. They proceeded to measure *CoCo* using SonarQube[2] to obtain the *CoCo* value for each source code snippet. This work resulted in a dataset, whose main characteristics are described in Tables 2 and 3, where: Did is the dataset identifier, SNo indicates the number of code snippets involved in the empirical study, while PNo and Demographic are the number and the type of participants, and Task indicates the activity through which understandability was assessed. The dataset was published in the supplemental material (Barón et al., 2020b) of the published study (Barón et al., 2020a).

By analyzing the dataset, Muñoz Barón et al. evaluated the association of *CoCo* with the measures of the various aspects of understandability: the time taken to understand a code snippet, the percentage of correctly answered comprehension questions on a code snippet, subjective ratings of a comprehension task, and physiological measures on the subjects engaged in understanding code.

Muñoz Barón et al. reported the correlation between *CoCo* and each of the mentioned aspects of understandability for each of the 10 experiments reported in the selected papers, as well as a summary obtained via meta-analysis. Muñoz Barón et al. concluded that *CoCo* correlates with the time it takes a developer to understand source code, with a combination of time and correctness, and with subjective ratings of understandability.

The work by Muñoz Barón et al. has two relevant merits. It is the first attempt at deriving an objective evaluation of the capability of *CoCo* in indicating practically relevant aspects of understandability. It also provided a valuable dataset that is available to researchers that would like to extend the research on *CoCo*. However, the work by Muñoz Barón et al. does not address the base claim underlying the introduction of *CoCo*. In fact, *CoCo* was proposed with the aim *"to remedy Cyclomatic Complexity's shortcomings and produce a measurement that more accurately reflects the relative difficulty of understanding, and therefore of maintaining methods, classes, and applications"* (Campbell, 2018a). Therefore, before embracing the use of *CoCo*, we need to understand whether *CoCo* is really correlated with understandability better than the measures that were proposed in the past for the same purpose (e.g., those listed in Table 1). This kind of verification was not undertaken by Muñoz Barón et al. hence we decided to carry out this investigation.

3.3. Related work on nesting measures

The issue of the impact of nesting on source code complexity was addressed in previous software measurement literature especially with the goal of overcoming some limitations of *McCC*. For instance, methods `firstMethod` and `secondMethod` in Section 3.1 above have the same *McCC*, but from an intuitive point of view, the first routine should be considered more "complex", i.e., more difficult to write or understand, than the second one.

A number of nesting-aware measures were proposed in the literature. We now concisely review a few of them. Howatt and Baker (1989) provided a formal definition of nesting that can be applied to structured and unstructured programming. This formal definition was used as a framework for several nesting-based measures, which we now list.

- Piwowarski (1982) introduced a complexity measure N as the sum of two terms. The first one is a modified version of *McCC*, in which all `switch` statements, regardless of the number of their branches, are counted as if they were `if` statements. The second one is the sum of the nesting levels at which predicates in the control structures are found. For instance, `firstMethod` in Section 3.1 has $N = 6$, while `secondMethod` has $N = 3$.

[2] https://www.sonarqube.org/

Table 2
Characteristics of the datasets retrieved from relevant studies.

Did	Ref	Language	SNo	PNo	Demographic	Task	Understandability aspects
1	Siegmund et al. (2012)	Java	23	41	Students	Calculate output	Time, correctness, rating
2	Peitek et al. (2020)	Java	12	28	Students	Calculate output	Time, physiological
3	Buse and Weimer (2010)	Java	100	120	Students	Rate snippet	Rating
4	Dolado et al. (2003)	C/C++	20	51	Prof. & Stud.	Answer questions	Time, correctness
5	Salvaneschi et al. (2014)	Scala	20	38	Students	Answer questions	Time, correctness
6	Scalabrino et al. (2021)	Java	50	63	Professionals	Rate and answer	Time, correctness, rating
7	Hofmeister et al. (2017)	C#	6	72	Professionals	Find bug	Time
8	Ajami et al. (2019)	JavaScript	40	220	Professionals	Calculate output	Time
9	Börstler and Paech (2016)	Java	30	104	Students	Rate and cloze test	Time, correctness, rating
10	Gopstein et al. (2017)	C/C++	126	73	Students	Calculate output	Time, correctness

Table 3
Characteristics of the datasets retrieved from relevant studies.

		Cognitive Complexity				
Did	Mean	Med	Min	Max	StDev	Understandability measures
1	3.26	3	0	9	2.43	`time` (time), `correct` (correctness), `confidence`, `difficulty` (rating)
2	2.50	2	0	6	1.88	`resp_time` (time), `BA31_ant`, `BA31_post`, `BA32` (physiological)
3	1.34	1	0	10	1.74	`perceived_readability` (rating)
4	2.55	2	0	8	2.35	`time_ta`, `time_tb`, (time), `correct_ta`, `correct_tb` (correctness)
5	2.55	2	0	7	1.82	`time` (time), `correct` (correctness)
6	8.56	7.5	0	46	8.53	`tnpu` (time), `au` (correctness), `pbu` (rating)
7	2.17	2	1	4	1.33	`first_impression`, `thinking_time`, `duration` (time)
8	6.50	4	1	14	3.6	`correct_and_wrong` (time)
9	8.00	4	0	16	6.81	`tr`, `ta` (time), `acc` (correctness), `r1`, `r2` (rating)
10	1.17	1	0	8	1.4	`duration`, (time), `correct` (correctness)

- Dunsmore and Gannon (1979) defined a measure based on the average control flow nesting of a program.
- Harrison and Magel introduced two measures for structured programming. The "Scope Number" (Harrison and Magel, 1981a) takes into account the nesting level of each control structure. The "Scope Ratio" (Harrison and Magel, 1981b) is the ratio of the number of nodes in a control flow graph to the "Scope Number".
- Chen (1978) defines an entropy-based complexity measure that also accounts for nesting of predicate nodes.

Also, Li (1987) introduces a measure similar to *CoCo*, in which each control structure is weighted according to its nesting level. Binary logical operators, instead, are not weighted according to the nesting level of the control structure they belong to.

4. The empirical study

Our empirical study is based on the same primary studies as the empirical study by Muñoz Barón et al. We concisely review them in Section 4.1. We describe how we obtained the data about these primary studies by complementing the data used by Muñoz Barón et al. in Section 4.2. The data analysis procedure we used is described in Section 4.3.

4.1. Primary studies on code understandability

Dolado et al. (2003) investigated whether side effects due to auto-increments (e.g., "`i++`" "`++i`") and auto-decrements (e.g., "`i--`" "`--i`") have an impact on program comprehension. They (1) built pairs of equivalent C code fragments with and without auto-increments and auto-decrements; (2) asked the same set of comprehension questions for both members of each pair; (3) measured the number of correct answers and the time spent in answering. The results showed that these side effects significantly reduced performance in comprehension-related tasks.

Buse and Weimer (2010) used a set of characteristics of Java code (e.g., the number of assignments, the number of keywords, the number of parentheses) to build readability classifiers. They checked whether the classifiers were statistically related with a two-valued ("more readable", "less readable") readability rating. They carried out an empirical study with 120 participants, each of whom evaluated the readability of 100 Java snippets, each of which consisted of precisely three consecutive simple statements. Readability evaluation was done by the participants on a 5-value ordinal readability scale, which was later split into the two-valued "more readable", "less readable" scale. The results found that some of the classifiers achieve fairly good values of F-measure (Van Rijsbergen, 1979) (around 0.8). The classifiers also appeared to be related with code defects, code churn, and self-reported stability. Börstler and Paech (2016) examined the role of method chains and code comments in software readability and comprehension through an empirical study that included 104 students with different degrees of programming experience. Readability and comprehension were captured by means of perceived readability, reading time, and performance on a simple cloze test (Osgood et al., 1954). The results show that code comments affect perceived readability, but not comprehension as measured by the accuracy of answers to cloze questions. There does not seem to be any impact of the presence of method chains on perceived readability or comprehension, contrary to conventional wisdom. Also, perceived readability does not appear to be related to comprehension.

Ajami et al. (2019) conducted an experiment involving 220 professional developers to investigate how some syntactical code structures influence code comprehension in terms of time and correctness. Among other results, they found that some programming control structures are harder than others, e.g., `for` loops are more difficult than `if` statements, loops counting down are slightly harder to understand than loops counting up.

Scalabrino et al. (2021) studied the existence of statistical associations between 121 code measures and perceived and actual code understandability in terms of time, correctness, and subjective rating, but none of these associations was at a medium or strong level. Using combinations of some of these measures improves the results, but not to the point that they are practically usable.

Gopstein et al. (2017) studied code patterns and identified atoms of confusion that usually confuse developers and might lead them to make mistakes. They empirically showed that these

patterns can produce a significant rate of misunderstanding against code without these patterns. Removing these patterns improved substantially the developers' ability to grasp the code.

Hofmeister et al. (2017) investigated the effects of identifiers naming styles (letters, abbreviations, words) on program comprehension. They conducted an empirical study with 72 professional C# developers and measured the time needed to find defects in code snippets. They found that using full words as identifiers (e.g., a variable spelled out as `first`) led to 19%-time improvement to find defects compared to using abbreviations (e.g., a variable `frs`) or single letters (e.g., a variable `g`). No statistically significant time difference was found between the use of abbreviations and single letters.

Salvaneschi et al. (2014) performed an empirical study that showed that Reactive Programming code was more understandable than Object-Oriented in terms of time (for performing tasks on the code) and correctness (in answering questions about the code).

Siegmund et al. (2012) conducted a controlled experiment on 41 participants to observe what happens inside the brain during program comprehension. Using functional Magnetic Resonance Imaging (fMRI), they identified the brain areas that are activated during the cognitive process needed for program comprehension (specifically, locating syntax errors in short code snippets). They observed the activation of brain regions that are responsible for functions related to bottom-up program comprehension, i.e., working memory, attention, and language processing.

Peitek et al. (2020) used fMRI scanner to measure program comprehension. They observed 28 participants who had the task of understanding 12 source code snippets during the fMRI. One of the Research Questions investigated whether source code complexity was related to concentration levels during bottom-up program comprehension. Specifically, the code measures chosen were: *LOC*, *McCC*, Halstead's "complexity" (probably Halstead Difficulty), and *DepDegree*, defined in Beyer and Fararooy (2010) as the number of dependency edges in the definition-use graph and believed to be related to readability and understandability. The data analysis checked whether these measures were linearly correlated with the average value across the participants of some physiological variables, so the dataset used was composed of 12 datapoints, one for each snippet. Reasonably high linear correlations were found with Halstead's "complexity" and *DepDegree*, but not with *LOC* or *McCC*. Overall, it appears that this avenue of research has potential for further developments.

It is worth noticing that, like Muñoz Barón et al. we used the datasets of each of these primary studies, regardless of the fact that the data associated with a study had been fully used in the corresponding publication. For instance, Gopstein et al. reported only about correctness results in their paper (Gopstein et al., 2017), but Muñoz Barón et al. were able to retrieve also timing data concerning the experiment by Barón et al. (2020b).

4.2. The dataset

The data we analyzed were obtained as follows.

- We first retrieved the supplemental material of the paper by Muñoz Barón et al. (2020b) and we used the `dataset.xlsx` spreadsheet where *CoCo* measures and understandability evaluations were recorded.
- We then retrieved the source code used in the studies selected by Muñoz Barón et al. whose data populate the `dataset.xlsx` file. In doing this, we used the links to the code provided in the supplemental material provided by Muñoz Barón et al.
- Finally, we used SourceMeter on the retrieved source code to obtain the code measures listed in Table 1, which we added to the `dataset.xlsx` file. The descriptive statistics of code measures are in Table 4.

Table 4
Descriptive statistics of code measures.

Measure	Mean	St.dev.	Median	Min	Max	Available for datasets
CoCo	3.34	4.94	2	0	46	all
LLOC	13	12	9	3	66	all
McCC	3.08	2.62	2	1	19	all
NLE	1.04	1.11	1	0	6	all
MI	60	51	80	0	136	1,2,3,6,9
HVOL	473	474	273	24	2613	1,2,3,6,9
HCPL	158	118	117	18	633	1,2,3,6,9

Our resulting dataset therefore contains, for each empirical study selected by Muñoz Barón et al. the value of the understandability aspects considered (time, correctness, rating, physiological) and *CoCo* along with the more traditional source code measures listed in Table 1 (or a subset thereof, in some cases).

Note that not all the considered measures were available for all datasets. In fact, the support provided by SourceMeter for the C family languages and for Javascript is not complete: MI, HVOL, and HCPL are not measured. Therefore, such measures were neither available for projects 4, 7 and 10 (which are written in C/C++ and C#) nor for project 8 (which is written in Javascript). In addition, SourceMeter does not support the Scala language at all. Therefore, we manually extracted some measures (namely LLOC, McCC and NLE), but neither Halstead measures nor the maintainability index for project 8, which is written in Scala.

Column "available for datasets" of Table 4 indicates the datasets for which each measure was available.

4.3. Method

To address the research questions, we built models that predict understandability aspects based on source code measures. In an exploratory phase, we tried building models using a few machine learning techniques, namely Support Vector Regression (SVR), Neural Networks (NN), and Random Forests (RF). In the great majority of cases, SVR provided more accurate predictions than the other techniques, so we adopted SVR as the technique to build models.

The details concerning the construction of SVR models are given in Appendix A.

To evaluate the performance of models, since the datasets are not very large (the biggest one contains 126 data points, as shown in Table 2), we opted for leave-one-out cross validation: the understandability of every snippet was evaluated via a model built using all the remaining snippets.

The accuracy of the obtained predictions was evaluated via MAR, which is an unbiased indicator, recommended by several authors (e.g., Shepperd and MacDonell, 2012)). Given a set of observations Y, the residual (or error) of the ith prediction $\hat{y}_i$ is $y_i - \hat{y}_i$, where y_i is the ith observation (i.e., the actual value of the considered understandability factor). The MAR is then computed as the mean of absolute residuals, as follows:

$$MAR = \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

Unfortunately, the MAR obtained for different datasets and different understandability factors are not comparable, because each factor in each dataset represents a different concept. This is evident when considering factor `time` in the dataset having Did = 1 and factor `time_ta` in the dataset having Did = 2. Even though

in both cases the measured aspect is the time taken to complete a task concerning code understanding, the differences of tasks, conditions, and performers make the measures not directly comparable. In fact, `time` in dataset 1 is in the [19.4, 360.2] range, with mean 90.5 and median 77.5, while `time_ta` in dataset 2 is in the [15.2, 54.7] range, with mean 29.4 and median 25.5.

This is the same situation you face when comparing two cars, that you test by several short trips entirely in town and several longer trips mainly on highway. If you need to evaluate a model that predicts fuel consumption, when comparing predictions, you have to take into account that the in-town trip is shorter and slower, while the country trip is longer and faster. Dividing all measures by the mean consumption observed for the corresponding type of trip is a way to normalize measures and make them comparable.

We make a similar normalization: to get error indicators that are comparable across datasets and understandability factors, we normalize them by dividing errors by the mean actual value of the measured aspect; i.e., we use the mean actual value as the unit of measure of each understandability aspect. Specifically, we proceed as follows: given a set of observations Y,

- The residual of the ith prediction $\hat{y}_i$ is $y_i - \hat{y}_i$, where y_i is the ith observation.
- The mean actual value $\bar{y}$ is $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$, where n is the number of observations in Y.
- We consider the ratio rr between absolute residuals and the mean of actuals: $rr_i = \frac{|y_i - \hat{y}_i|}{\bar{y}}$.
- Then, we compute MR, the mean of rr, as follows:

$$MR = \frac{1}{n}\sum_{i=1}^{n} rr_i = \frac{1}{n}\sum_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{\bar{y}} = \frac{1}{\bar{y}}\frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i| = \frac{MAR}{\bar{y}}$$

In this way, we get MR values that are comparable across datasets. For instance, the MAR obtained for dataset Did = 1 when predicting `time` is 28.78, and the MAR obtained for dataset Did = 2 when predicting `time_ta` is 7.73. Although $7.73 \ll 28.78$, we cannot state that the former prediction is much more accurate than the latter, since `time` has a different meaning (and is obtained under different conditions) with respect to `time_ta`. By considering that the mean `time` in dataset Did = 1 is 90.5 and the mean `time_ta` in dataset Did = 2 is 29.4, we get MR = 0.31 for `time` and MR = 0.26 for `time_ta`: the prediction of `time_ta` is more accurate than the prediction of `time`, but to a much lesser extent than indicated by MAR.

Unlike MMRE, i.e., the Mean Magnitude of Relative Errors, defined as $\frac{1}{n}\sum_{i=1}^{n} \frac{|y_i - \hat{y}_i|}{y_i}$, MR is not biased, since in the computation of MR the absolute residuals of a given dataset are all divided by the same number (the mean value of the considered measure in that dataset).

Now, we have to consider that our data tend to be skewed, since the experiments that produced the data involved mainly small and simple pieces of code. Therefore, to provide a more adequate evaluation of the situation, we also computed indicators that are less sensitive to skewness than MAR and MR. Specifically we computed the median of absolute residuals (MdAR) and the ratio MdR between MdAR and the median absolute value of the considered understandability factor. The evaluations based on MdAR and MdR are reported in Appendix B. They confirm the evaluations based on MAR and MR.

We must also consider that, when considering a specific understandability factor and a specific dataset, we can find that two measures (or sets of measures) obtain very close MAR values. In those cases, it is critical to evaluate to what extent a prediction is better than the other one. To perform this type of evaluation, we also computed the effect size on absolute residuals. The effect size was computed via Hedges's g, i.e., via Cohen's d statistics (Cohen, 2013) with Hedges correction (Hedges and Olkin, 2014).

To answer RQ1, we built and compared models using one independent variable, and compared the models based on *CoCo* with the models based on traditional code measures.

To answer RQ2, we built models using as many independent variables as possible without overfitting, and compared the models that use *CoCo* as an independent variables with the models that do not, i.e., those based only on traditional code measures. In this way we evaluated whether using *CoCo* improves the predictability of understandability aspects.

5. Results

5.1. Results concerning RQ1

In this section, we report about the models based on a single measure, which we built to answer research question RQ1.

5.1.1. Results concerning the time required to understand code

We obtained SVR models of the time required to understand code for all the datasets, and for all the available code measures, with one exception: for dataset 4 (task a) no model using NLE could be found.

Table 5 illustrates the performance values of the models for predicting the time required to understand software code. Column Did provides the identifier of the dataset and the name of the understandability factor in parentheses, when more than one factor was evaluated from the same dataset. The other columns provide MAR and MR (as a percentage) for each of the considered measures.

"NoData" indicates that the software measure was not available, so no model based on it could be built. "NoModel", instead, denotes that no model could be built, even though the data for the corresponding measure were available. This is the case for measure NLE: no model based on NLE could be found for datasets 4(ta), 5 and 7 because NLE is very close to being a constant in these datasets. Table 5 shows that the ability of code measure in predicting the time needed to understand code is not very good, in general. For instance, for dataset 1 we have that even the best predictor has MR = 32%, indicating that the average absolute error is close to one third of the average time actually needed to understand the code.

Table 5 shows that there is no measure that supports the best predictions across the datasets. It is also apparent that the various measures achieve similar accuracy, except in rare cases (e.g., LLOC does not work well with dataset 4).

To evaluate to what extent a model is more accurate than others, we computed the effect size using Hedges' g (Rosenthal et al., 1994), as described in Section 4.3.

Table 6 shows the results we obtained. Every cell of the table represents a comparison with *CoCo* absolute residuals: the minus sign indicates that *CoCo* performed better, while the plus sign reports that *CoCo* performs worse than the concerned measure. Gray cells indicate that the measure was not available for the considered dataset, hence no comparison was possible.

To make the table easier to read, we report the interpretation suggested by Cohen (2013), rather than the sheer value of g:

- $|g| < 0.2$ indicates a negligible effect size: the cell is empty.
- $0.2 \leq |g| < 0.5$ indicates a small effect size: the cell contains '*'.
- $0.5 \leq |g| < 0.8$ indicates a medium effect size: the cell contains '**'.
- $0.8 \leq |g|$ indicates a large effect size: the cell contains '***'.

Table 5
Accuracy of models based on source code measures, when predicting the time needed to understand code, evaluated via MAR and MR.

Did	CoCo	McCC	LLOC	NLE	MI	HVOL	HCPL
1	28.78 (32%)	37.33 (41%)	38.67 (43%)	42.00 (46%)	43.15 (48%)	30.76 (34%)	45.80 (51%)
2	7.73 (26%)	8.64 (29%)	6.32 (22%)	9.93 (34%)	7.74 (26%)	6.52 (22%)	4.99 (17%)
4 (ta)	17.71 (32%)	15.17 (28%)	25.69 (47%)	NoModel	NoData	NoData	NoData
4 (tb)	7.02 (13%)	11.95 (22%)	35.15 (66%)	10.34 (19%)	NoData	NoData	NoData
5	35.92 (26%)	43.87 (32%)	36.05 (26%)	NoModel	NoData	NoData	NoData
6	44.72 (50%)	57.62 (65%)	45.73 (52%)	45.20 (51%)	48.33 (55%)	49.11 (55%)	47.80 (54%)
7 (first imp.)	15.39 (16%)	16.13 (17%)	15.69 (16%)	NoModel	NoData	NoData	NoData
7 (thinking)	8.51 (27%)	9.01 (28%)	8.73 (28%)	NoModel	NoData	NoData	NoData
7 (duration)	25.08 (18%)	27.06 (20%)	24.20 (17%)	NoModel	NoData	NoData	NoData
8	58.00 (27%)	46.26 (21%)	49.43 (23%)	54.58 (25%)	NoData	NoData	NoData
9 (tr)	24.77 (21%)	24.67 (21%)	20.85 (18%)	23.70 (20%)	19.23 (16%)	24.40 (21%)	23.64 (20%)
9 (ta)	17.09 (18%)	15.89 (17%)	19.23 (21%)	15.88 (17%)	21.70 (23%)	22.40 (24%)	21.69 (23%)
10	7.10 (34%)	6.88 (33%)	6.67 (32%)	6.68 (32%)	NoData	NoData	NoData

Table 6
Effect size for time models, computed via Hedges' g.

Did	McCC	LLOC	NLE	MI	HVOL	HCPL
1		$-^{*}$	$-^{*}$	$-^{*}$		$-^{*}$
2		$+^{*}$	$-^{*}$		$+^{*}$	$+^{**}$
4 (ta)	$+^{*}$	$-^{*}$				
4 (tb)	$-^{*}$	$-^{***}$	$-^{*}$			
5	$-^{*}$					
6	$-^{*}$					
7 (first imp.)						
7 (thinking)						
7 (duration)						
8	$+^{*}$					
9 (tr)				$+^{*}$		
9 (ta)				$-^{*}$	$-^{*}$	$-^{*}$
10						

In practice, Table 6 confirms that no variable appears consistently better than the others in predicting the time needed to understand code. In addition, the effect size is generally negligible or small. Only HCPL achieves a medium positive effect size over *CoCo* in dataset 2, and LLOC shows a large negative effect size with respect to *CoCo* in dataset 4 (task b). It is interesting to note that for two datasets (7 and 10) *CoCo* appears to provide a performance level that is only (at best) marginally different from the other code measures.

Finally, we can note that in only one case out of 13, namely, for dataset 4 and task b, *CoCo* outperforms all the other available code measures. In the remaining 12 cases it is always possible to use a model that does not use *CoCo* and performs not worse than the model using *CoCo*.

5.1.2. Results concerning understandability ratings

We obtained models of the understandability rating for datasets 1, 3, 6, and 9, that is, for all the datasets that provide understandability rating data.

The MAR and MR values for each dataset and each variable are given in Table 7. Also in this case, there is no measure that supports the best predictions across all datasets, although HCPL always gets close to best performance. In general, the achieved accuracy is better than for the time taken to understand code (Table 5).

Table 8 shows the results obtained from the evaluation of the effect size. It is easy to see that *CoCo* performs approximately as accurately as traditional measures: in 31 comparisons out of 36, the effect size is negligible, and in the remaining 5 cases it is small.

5.1.3. Results concerning correctness of code understanding

We obtained models of the correctness of understanding for all the datasets that provide understanding correctness data, with one exception: for dataset 4 (task a) no model using NLE could be found.

The MAR values for each dataset and each variable are given in Table 9.

As in the former cases, no measure provides the most accurate predictions across the datasets. Noticeably, *CoCo* never achieves the best results, although it gets close to the best, like several other measures. In fact, all measures achieve very similar accuracy.

Table 10 shows the results obtained from the evaluation of the effect size. It is easy to see that *CoCo* performs approximately as traditional measures in all but two cases.

5.1.4. Results concerning physiological aspects of code understanding

Only dataset 2 provides data concerning the physiological aspects of code understanding. It supported the derivation of models for all the understandability aspects it covers.

The MAR and MR values for each model are given in Table 11. Also in this case there is no measure that always achieves the best accuracy. Table 12 shows the results obtained from the evaluation of the effect size. With respect to the previous cases, the difference in accuracy appears larger, with two medium and one large effect sizes. However, in all cases there is at least one traditional code measure whose performance is not appreciably different form *CoCo*'s.

5.2. Results concerning RQ2

In this section, we report about the models based on multiple measures, which we built to answer research question RQ2.

To avoid overfitting, we used a maximum number of measures equal to $\lfloor \frac{n}{10} \rfloor$, where n is the number of data points in the considered dataset. For instance, dataset 1 includes 23 data points, hence we built models with no more than two independent variables for dataset 1.

We built (1) models using all the available measures (including *CoCo*) and (2) models using only traditional measures. The idea is that if models that use *CoCo* do not get appreciably better performance than models that do not use *CoCo*, the contribution of *CoCo* to models' performances is negligible.

Table 13 describes the best multivariate models that we found for each dataset and understandability aspect. In general, the best model found for a dataset and understandability aspect uses only a subset of the measures; in some cases, the best model happens to use just one measure. For some dataset and aspect, the best models both with and without *CoCo* were based on a single measure: these models are not reported here, since they were already shown in Section 5.1. For each dataset and understandability aspect, Table 13 reports

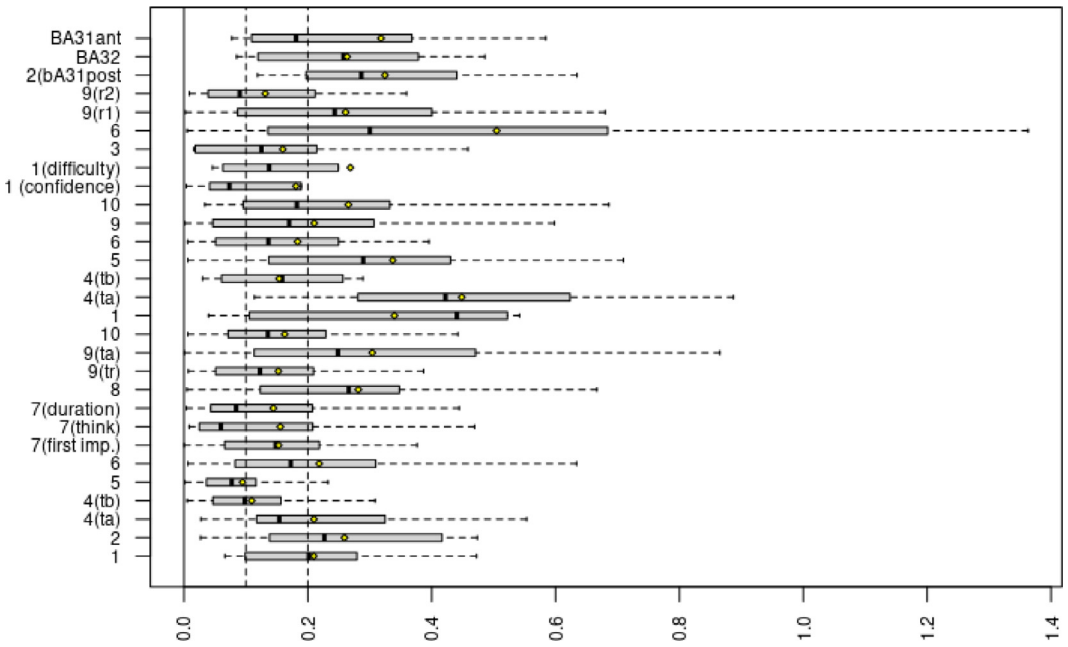

Fig. 1. Boxplots of MR (the ratio between absolute residuals and mean actuals), for all the models using *CoCo* as the independent variable (outliers not shown).

Table 7
Accuracy of models based on source code measures, when predicting understandability rating, evaluated via MAR and MR.

Did	CoCo	McCC	LLOC	NLE	MI	HVOL	HCPL
1 (confidence)	0.44 (14%)	0.46 (15%)	0.34 (11%)	0.44 (14%)	0.43 (14%)	0.41 (14%)	0.34 (11%)
1 (difficulty)	0.45 (16%)	0.42 (15%)	0.38 (13%)	0.41 (14%)	0.53 (18%)	0.37 (13%)	0.34 (12%)
3	0.50 (15%)	0.55 (17%)	0.49 (15%)	0.51 (16%)	0.50 (15%)	0.41 (13%)	0.42 (13%)
6	0.15 (22%)	0.14 (21%)	0.15 (22%)	0.16 (22%)	0.17 (24%)	0.17 (25%)	0.16 (23%)
9 (r1)	0.26 (9%)	0.24 (9%)	0.24 (9%)	0.24 (9%)	0.25 (9%)	0.26 (10%)	0.26 (9%)
9 (r2)	0.28 (11%)	0.28 (11%)	0.28 (11%)	0.29 (11%)	0.28 (11%)	0.29 (11%)	0.30 (12%)

Table 8
Effect size for understandability rating models.

Did	McCC	LLOC	NLE	MI	HVOL	HCPL
1 (confidence)		+*				+*
1 (difficulty)						+*
3					+*	+*
6						
9 (r1)						
9 (r2)						

- The variables that support the best model using *CoCo*;
- The MAR and MR of the best model that uses *CoCo*;
- The variables that support the best model that does not use *CoCo*;
- The MAR and MR of the best model that does not use *CoCo*.

Table 13 shows that for each dataset and understandability aspect, the best models achieve extremely similar accuracy. The effect size indicates negligible accuracy differences for all dataset and aspect pairs.

6. Answers to research questions

6.1. Answer to research question RQ1

RQ1 ("How well is *CoCo* correlated with software understandability, also in comparison with 'traditional' measures?") actually involves two issues:

RQ1.1 How well is *CoCo* correlated with software understandability in absolute terms, i.e., does *CoCo* provide reliable indications concerning understandability?

RQ1.2 Is *CoCo* any better at predicting code understandability than other measures?

To answer RQ1.1, we can consider the models described in Section 5.1 and for each one compute *rr*, the ratio between its absolute residuals and the mean of actuals. Fig. 1 shows the boxplots of *rr*. The yellow diamonds indicate the mean values, i.e., MR. To keep the figure readable, outliers have been omitted from the boxplots.

Although it is not easy to set a threshold that partitions "good predictions" and "bad" predictions, it is safe to assume that *rr* below 10% indicate reliable results, while *rr* above 20% indicate not sufficiently reliable ones. Therefore, we represent these thresholds in Fig. 1 as dashed lines.

It can be observed that in 11 cases out of 29 the median is above the 20% threshold, while in only 6 cases it is below the 10% threshold. In addition, except for time models 4(tb) and 5, at least 25% of the predictions have MR greater than 20%. Accordingly, we can conclude that *CoCo* appears correlated to understandability aspects, but not so well as to yield reliable predictions concerning readability aspects.

Let us now address RQ1.2. To this end, we consider both the data given in Tables 5, 7, 9, and 11 as well as the effect size data given in Tables 6, 8, 10, and 12.

The results reported in the mentioned tables indicate a large variability: no measure seems to perform consistently better than

Table 9
Accuracy of models based on source code measures, when predicting understandability correctness, evaluated via MAR and MR.

Did	CoCo	McCC	LLOC	NLE	MI	HVOL	HCPL
1	0.25 (15%)	0.27 (17%)	0.20 (13%)	0.26 (16%)	0.22 (14%)	0.22 (14%)	0.23 (14%)
4 (ta)	0.33 (45%)	0.31 (42%)	0.18 (25%)	NoModel	NoData	NoData	NoData
4 (tb)	0.19 (34%)	0.18 (33%)	0.22 (39%)	0.19 (34%)	NoData	NoData	NoData
5	0.12 (16%)	0.12 (17%)	0.14 (19%)	NoModel	NoData	NoData	NoData
6	0.14 (30%)	0.14 (31%)	0.14 (32%)	0.13 (30%)	0.15 (34%)	0.15 (34%)	0.15 (33%)
9	0.07 (15%)	0.07 (16%)	0.07 (15%)	0.07 (17%)	0.06 (15%)	0.07 (16%)	0.06 (15%)
10	0.20 (28%)	0.20 (28%)	0.19 (27%)	0.20 (28%)	NoData	NoData	NoData

Table 10
Effect size for understanding correctness models.

Did	McCC	LLOC	NLE	MI	HVOL	HCPL
1		+*				
4 (ta)		+***				
4 (tb)						
5						
6						
9						
10						

the other ones. In addition, all measures provide similar performances on most datasets.

The evaluation of the effect size shows that in the vast majority of cases *CoCo* provides performances that are extremely close to other measures'. In the remaining cases, we have that

- In 23 cases out 29 the differences in performance by *CoCo* and McCC are negligible. In 4 cases out of 6, *CoCo* performs slightly better than McCC (effect size small), while in the remaining two cases McCC performs better (effect size small).
- In 20 cases out 29, the differences in performance by *CoCo* and LLOC are negligible. In the remaining 9 cases, *CoCo* performs better 4 times (3 times with small and once with large effect size), while in the other 5 cases LLOC performs better (4 times with small and once with large effect size).
- In 8 cases out 17, the differences in performance by *CoCo* and HPCL are negligible. In the remaining 9 cases *CoCo* performs better 3 times (2 times with small and once with medium effect size), while in the 6 cases HCPL performs better (3 times with small, two with medium and once with large effect size).

So, the answer to RQ1.2 is that *CoCo* appears to provide a level of correlation with understandability aspects that is quite similar to most of the other measures'. There is no evidence that *CoCo* can be considered a better predictor of understandability than other measures. On the contrary, a few measures that were defined long ago, namely *LLOC* and Halstead *HCPL*, appeared to perform marginally better than *CoCo*.

6.2. Answer to research question RQ2

RQ2 asks "Using *CoCo*, is it possible to build better predictors of code understandability than using only traditional source code measures?"

To answer RQ2, we proceeded to search, for each dataset and understandability factor, the best model that does not use *CoCo* and the best model that uses *CoCo*. Then we checked whether using *CoCo* increases our ability to predict readability aspects with increased accuracy.

As described in Section 5.2, for eleven datasets and understandability factors we found multivariate models (see Table 13) with improved performance over the univariate ones (described in Section 5.1). Table 13 shows that models using *CoCo* have a level of accuracy that is extremely close to models not using *CoCo*. Therefore, the multivariate models do not change the conclusions we reached when considering RQ1.2: using *CoCo*, it is possible to build predictors of code understandability that are neither substantially better nor substantially worse than predictors that do not use *CoCo*.

In summary, according to our data, *CoCo* appears neither better nor worse than well-established static code measures, with respect to evaluating code understandability.

6.3. Additional observations

CoCo, as well as other measures, appear to correlate fairly well with some understandability factor in some cases. For instance, Table 9 shows that both *CoCo* and other measures can predict understanding correctness for datasets 1, 5 and 9, while performances are definitely poor with datasets 4, 6, and 10.

This observed performance variability might be in part due to two reasons.

- In a relatively large number of datasets, measures mildly correlated to understandability can be used to build fairly acceptable models by chance every now and then, though in an inconsistent way. The higher the correlation, the higher the likelihood that these measures can consistently be used for understandability models.
- Given a dataset and an understandability aspect, most measures tend to achieve similar estimation accuracy. This phenomenon is probably due to the fact that most code measures appear correlated with each other (Lavazza, 2020).

At any rate, these are preliminary explanations that may be further refined by future studies with a larger number of independent variables and with a larger number of datasets, which may also provide indications for additional reasons.

7. Threats to validity

As already mentioned, we reused the data collected by Barón et al. (2020b). Therefore, the same threats reported by Muñoz Barón et al. apply (Barón et al., 2020a). They can be summarized as follows.

- In the search for reusable data concerning experiments on code understandability, some relevant study may have been missed.
- Some code snippets had to be marginally altered in order to free them of syntax errors and dependency issues so that *CoCo* could be calculated automatically.
- Many of the code snippets featured low values of *CoCo*.

Table 11
Accuracy of models based on source code measures, when predicting Physiological Aspects of understandability, evaluated via MAR and MR.

Did	CoCo	McCC	LLOC	NLE	MI	HVOL	HCPL
2 (BA31ant)	0.13 (21%)	0.14 (22%)	0.13 (22%)	0.14 (23%)	0.14 (23%)	0.12 (20%)	0.07 (11%)
2 (BA31post)	0.13 (21%)	0.16 (25%)	0.17 (27%)	0.13 (20%)	0.16 (26%)	0.20 (31%)	0.20 (31%)
2 (BA32)	0.19 (26%)	0.20 (27%)	0.17 (22%)	0.19 (26%)	0.19 (25%)	0.17 (23%)	0.12 (17%)

Table 12
Effect size for models of physiological aspects of understandability.

Did	McCC	LLOC	NLE	MI	HVOL	HCPL
2 (BA31ant)						$+^{***}$
2 (BA31post)	$-^{*}$	$-^{*}$		$-^{*}$	$-^{**}$	$-^{**}$
2 (BA32)		$+^{*}$				$+^{**}$

Concerning our extension of the work by Muñoz Barón et al. we also had to slightly modify the code in order to make it measurable by SourceMeter. However, just as Muñoz Barón et al. did, we took care not to alter the code properties that were going to be measured.

The estimation of software understandability requires that models be built (Morasca, 2009), but building the most accurate understandability models is not the goal of our investigation, nor was it of the previous study by Barón et al. (2020a), nor of the claim about the relationship between *CoCo* and understandability made in Campbell (2018a). So, it is possible that some other model building technique yields models that are more accurate than those illustrated in this paper; nonetheless, it is quite unlikely that in such models the considered code measures behave in a significantly different way than described here.

Both of our RQs – as well as the goal of the investigation in Barón et al. (2020a) – are about finding correlations between some source code measures (described in Section 3) and one or more understandability measures. Since understandability is an external software attribute, it must be borne in mind that it is unlikely that models based on code measures alone can be practically used for understandability prediction/estimation purposes, as they do not use any information about how the software code is dealt with by its "environment". Thus, this kind of correlational studies serve the purpose of finding possible predictors for understandability, and provide evidence for the potential usefulness of a source code measure.

8. Conclusions and future work

Being able to find statistical correlations between code understandability and source code measures would be greatly beneficial for the software development process, which involves a great deal of activities involving program comprehension (Minelli et al., 2015; Xia et al., 2017).

In 2018, a new measure, named "Cognitive Complexity" (*CoCo*) was proposed, with the purpose of providing code-based indications of code understandability that are more reliable than those provided by previously proposed code measures, such as *McCC* (McCabe, 1976) or maintainability indices (Oman and Hagemeister, 1992; Heitlager et al., 2007).

A first objective validation of *CoCo* was performed by Barón et al. (2020a). They reported the levels of association between *CoCo* and several aspects of understandability, based on data retrieved from previously published studies. They concluded that *CoCo* is moderately well associated with the time it takes a developer to understand source code, with a combination of time and correctness, and with subjective ratings of understandability.

The work by Muñoz Barón et al. was a necessary first step and provided the first empirical-based assessment of *CoCo*. However, the evidence reported by Muñoz Barón et al. does not address the claim underlying the definition of *CoCo* ("*a new metric that breaks from the use of mathematical models to evaluate code in order to remedy Cyclomatic Complexity's shortcomings and produce a measurement that more accurately reflects the relative difficulty of understanding, and therefore of maintaining methods, classes, and applications.*" Campbell, 2018a). Our paper aims to provide evidence about this underlying claim of *CoCo*.

To this end, we extended the work by Muñoz Barón et al. as an *in vitro* study to check whether the claim that *CoCo* provides advantages over other static measures, notably McCabe's Cyclomatic Complexity, was supported by evidence. We have thus studied the correlation between code understandability and several well-established source code measures. To this end, we reused the same data already used by Muñoz Barón et al. and built Support Vector Regression models of understandability aspects, based on source code measures.

Our results show that no code measure is consistently better correlated with code understandability than other measures. Specifically, *CoCo* does not appear to fulfill the promise of being a significant improvement with respect to previously proposed measures, as far as code understandability is concerned. Actually, *CoCo* does not seem to perform markedly worse or better than "traditional" source code measures. Noticeably, this conclusion applies to other measures as well: for instance, Halstead's HVOL does not seem to provide any advantages over other measures.

We expect that the results reported in this paper will be useful for both practitioners and researchers. Practitioners already using or considering to use *CoCo* may want to take into account the results provided by *CoCo* with a different degree of confidence, based on the evidence of our paper. Researchers may be interested in carrying out more empirical studies by using the results by Muñoz Barón et al. and the ones we provide. They may also come up with a different measure that leverages on the ideas underlying *CoCo* and other measures.

The code measures that supported both the study by Barón et al. (2020a) and ours concern mostly small code snippets. Code understanding performed during maintenance activities usually deals with bigger pieces of code. Thus, future work will encompass the following activities needed to generalize the results reported in papers (including this one) dealing with code understandability.

- It will be necessary to carry our empirical studies based on software code with bigger size and complexity.
- The association of several more source code measures with understandability will be investigated.
- Based on the results of the previous items, understandability models will be built, to support code understandability and maintenance by practitioners. Such models will be based not only on code measures, but also on measures of the "environment", including the professionals who have to understand the code and the goals of code understanding.
- Understandability will be investigated at higher abstraction levels than the detailed intra-method level. For instance, we will study the degree to which the coupling between two classes may influence the understandability of either class.

Table 13
Accuracy of best models using multiple variables, evaluated via MAR and MR.

aspect	Did	variables	MAR (MR)	variables	MAR (MR)
timing	6	*CoCo*, McCC, LLOC	40 (46%)	McCC, NLE	41 (47%)
	9 (tr)	*CoCo*, MI	22 (19%)	MI	19 (16%)
	9 (ta)	*CoCo*, McCC	16 (17%)	McCC, NLE	16 (17%)
	10	*CoCo*, McCC, LLOC	6.4 (30%)	LLOC, NLE, MI	6.6 (31%)
rating	3	*CoCo*, LLOC, MI	0.37 (11%)	McCC, LLOC, NLE, MI, HCPL	0.37 (11%)
	6	*CoCo*, McCC, HVOL, HCPL	0.15 (21%)	McCC	0.14 (21%)
	9 (r1)	*CoCo*, MI	0.24 (9%)	McCC, HCPL	0.22 (8%)
correct.	6	*CoCo*, McCC, NLE, HVOL	0.13 (29%)	McCC, NLE, HVOL	0.13 (29%)
	9	*CoCo*, HCPL	0.05 (12%)	NLE, HVOL	0.05 (12%)
	10	*CoCo*, NLE, MI	0.19 (26%)	LLOC	0.19 (27%)

CRediT authorship contribution statement

Luigi Lavazza: Conceptualization, Data curation, Investigation, Software, Supervision, Writing – original draft. **Abedallah Zaid Abualkishik:** Data curation, Writing – review & editing. **Geng Liu:** Data curation, Writing – review & editing. **Sandro Morasca:** Conceptualization, Methodology, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work has been partially supported by the "Fondo di ricerca d'Ateneo of the Università degli Studi dell'Insubria, Italy".

Appendix A. Details on the construction of SVR models

Data analysis was carried out using the R programming language and environment (R. core team, 2015). Specifically, we used the `e1071` library (https://cran.r-project.org/web/packages/e1071/index.html).

Since the used datasets contain predictors and a dependent variable, and values are all continuous, we chose a supervised and computationally not demanding method like SVR. Also the small size of the datasets was a criterion for choosing a robust approach like SVR. Finally, considering the problem at hand, we used a radial kernel.

To build models, a fundamental step was the configuration of the model with proper parameters (i.e., the so-called hyperparameters of the model). To this end, we exploited the `tune.svm` function of the `e1071` library. The `tune.svm` function was designed to find the best set of parameters for the data in a ranged or full parameter space for each parameter; we used this function passing a proper hyperparameter range for each tuning parameter:

cost is a regularization parameter used when transforming mathematically the problem into a Lagrangian formulation. We provided the tuning function with the $[2^{-3}, 2^{6}]$ range.

epsilon is the margin of tolerance for not penalizing errors. We provided the tuning function with the set of values {0.1, 0.01, 0.001}.

gamma controls the distance of the influence of a single training point. Low (respectively, large) values of gamma indicate a large (respectively, small) similarity radius which results in more (respectively, fewer) points being grouped together. We provided the tuning function with the $[2^{-1}, 2^{3}]$ range, which does not include large gamma values that could cause overfitting.

In addition, the `tune.svm` function has a `tune.control` argument, which enables the choice of common parameters like the sampling method, the size of the bootstrap samples, the returning of the error measure, and the returning of the performance of all the parameters combined at each tuning iteration. Among the `tune.control` arguments, `cross` allows the programmer to instruct the `tuning` function to look for the best parameters via an internal `cross`-fold cross validation: we set `cross = 5`.

Since the `tune.svm` function explores only a subset of the parameters space, we executed it ten times for each dataset, computing the resulting MAR; we then selected the parameters that obtained the lowest MAR.

Appendix B. Evaluations based on medians

In this appendix, we provide evaluations of understandability predictions based on MdAR and MdR. These evaluations are meant to complement those using MAR and MR shown above.

B.1. Evaluations concerning the time required to understand code

Table B.14 illustrates the accuracy of the models by providing the MdAR and MdR of the predictions. It is easy to see that Table B.14 confirms the results in Table 5 in that no code measure appears a consistently better predictor than other measures.

B.2. Evaluations concerning understandability ratings

Table B.15 illustrates the accuracy of the models by providing the MdAR and MdR of the predictions. It is easy to see that Table B.15 confirms the results in Table 7 in that no code measure appears a consistently better predictor than other measures.

B.3. Evaluations concerning correctness of code understanding

Table B.16 illustrates the accuracy of the models by providing the MdAR and MdR of the predictions. It is easy to see that also in this case the evaluations based on medians confirm the evaluations based on means, and there is no code measure that consistently outperforms the other ones.

B.4. Evaluations concerning physiological aspects of code understanding

Table B.17 illustrates the accuracy of the models by providing the MdAR and MdR of the predictions. It is easy to see that Table B.17 confirms the results in Table 11 in that no code measure appears a consistently better predictor than other measures.

Table B.14
Accuracy of models based on source code measures, when predicting the time needed to understand code, evaluated via MdAR and MdR.

Did	CoCo	McCC	LLOC	NLE	MI	HVOL	HCPL
1	16.38 (21%)	20.80 (27%)	21.10 (27%)	26.30 (34%)	30.53 (39%)	11.49 (15%)	35.56 (46%)
2	7.57 (30%)	6.97 (27%)	5.73 (23%)	7.99 (31%)	5.89 (23%)	4.68 (18%)	4.13 (16%)
4 (ta)	15.63 (36%)	12.67 (29%)	14.75 (34%)	NoModel	NoData	NoData	NoData
4 (tb)	4.82 (13%)	9.58 (26%)	27.90 (76%)	8.33 (23%)	NoData	NoData	NoData
5	33.48 (24%)	40.00 (29%)	31.12 (23%)	NoModel	NoData	NoData	NoData
6	26.59 (42%)	37.47 (59%)	26.44 (41%)	29.54 (46%)	35.83 (56%)	34.69 (54%)	29.87 (47%)
7 (first imp.)	12.06 (13%)	12.87 (14%)	12.36 (14%)	NoModel	NoData	NoData	NoData
7 (thinking)	4.35 (16%)	4.36 (16%)	4.35 (16%)	NoModel	NoData	NoData	NoData
7 (duration)	10.18 (8%)	10.11 (8%)	10.20 (8%)	NoModel	NoData	NoData	NoData
8	39.84 (20%)	34.50 (17%)	34.99 (18%)	40.57 (21%)	NoData	NoData	NoData
9 (tr)	20.00 (17%)	19.10 (16%)	9.47 (8%)	19.09 (16%)	11.63 (10%)	15.59 (13%)	16.29 (14%)
9 (ta)	12.70 (14%)	10.82 (12%)	14.03 (16%)	10.49 (12%)	16.89 (19%)	18.32 (21%)	16.24 (18%)
10	6.10 (31%)	5.83 (29%)	5.86 (29%)	5.16 (26%)	NoData	NoData	NoData

Table B.15
Accuracy of models based on source code measures, when predicting understandability rating, evaluated via MdAR and MdR.

Did	CoCo	McCC	LLOC	NLE	MI	HVOL	HCPL
1 (confidence)	0.26 (8%)	0.28 (9%)	0.24 (7%)	0.24 (7%)	0.25 (8%)	0.27 (8%)	0.18 (6%)
1 (difficulty)	0.17 (5%)	0.17 (5%)	0.30 (10%)	0.37 (12%)	0.52 (17%)	0.30 (10%)	0.24 (8%)
3	0.49 (14%)	0.54 (16%)	0.41 (12%)	0.48 (14%)	0.42 (12%)	0.34 (10%)	0.39 (11%)
6	0.12 (18%)	0.11 (17%)	0.14 (21%)	0.13 (20%)	0.17 (26%)	0.16 (23%)	0.12 (18%)
9 (r1)	0.21 (8%)	0.20 (7%)	0.21 (8%)	0.19 (7%)	0.20 (7%)	0.25 (9%)	0.26 (10%)
9 (r2)	0.25 (10%)	0.28 (11%)	0.25 (10%)	0.27 (11%)	0.21 (8%)	0.24 (9%)	0.30 (12%)

Table B.16
Accuracy of models based on source code measures, when predicting understandability correctness, evaluated via MdAR and MdR.

Did	CoCo	McCC	LLOC	NLE	MI	HVOL	HCPL
1	0.25 (16%)	0.20 (12%)	0.13 (8%)	0.27 (16%)	0.17 (11%)	0.12 (7%)	0.17 (10%)
4 (ta)	0.31 (38%)	0.31 (38%)	0.15 (18%)	NoModel	NoData	NoData	NoData
4 (tb)	0.25 (46%)	0.21 (40%)	0.21 (40%)	0.23 (43%)	NoData	NoData	NoData
5	0.10 (13%)	0.10 (13%)	0.10 (13%)	NoModel	NoData	NoData	NoData
6	0.11 (26%)	0.11 (25%)	0.12 (28%)	0.11 (26%)	0.12 (28%)	0.14 (32%)	0.14 (32%)
9	0.05 (13%)	0.06 (15%)	0.06 (14%)	0.07 (17%)	0.06 (15%)	0.06 (13%)	0.06 (15%)
10	0.19 (25%)	0.19 (25%)	0.18 (24%)	0.17 (23%)	NoData	NoData	NoData

Table B.17
Accuracy of models based on source code measures, when predicting Physiological Aspects of understandability, evaluated via MdAR and MdR.

Did	CoCo	McCC	LLOC	NLE	MI	HVOL	HCPL
2(BA31ant)	0.12 (20%)	0.16 (27%)	0.12 (19%)	0.16 (26%)	0.14 (23%)	0.11 (18%)	0.04 (7%)
2(BA31post)	0.10 (15%)	0.12 (19%)	0.17 (27%)	0.14 (21%)	0.12 (18%)	0.20 (32%)	0.19 (30%)
2(BA32)	0.17 (22%)	0.16 (21%)	0.18 (23%)	0.18 (24%)	0.14 (18%)	0.12 (15%)	0.09 (11%)

Table B.18
Accuracy of best models using multiple variables, evaluated via MdAR and MdR.

aspect	Did	variables	MdAR (MdR)	variables	MdAR (MdR)
timing	6	*CoCo*, NLE	20 (32%)	McCC, MI	21 (33%)
	9 (tr)	*CoCo*, MI	13 (11%)	LLOC	9 (8%)
	9 (ta)	*CoCo*, NLE	11 (12%)	NLE	10 (12%)
	10	*CoCo*, LLOC, MI	5.5 (28%)	NLE	5.2 (26%)
rating	3	*CoCo*, LLOC, NLE, MI, HCPL	0.27 (8%)	NLE, HCPL	0.27 (8%)
	6	*CoCo*, McCC, LLOC, HCPL	0.11 (16%)	McCC	0.11 (17%)
	9 (r1)	*CoCo*, MI	0.17 (6%)	LLOC, HVOL	0.17 (6%)
	9 (r2)	*CoCo*, LLOC	0.24 (9%)	MI	0.21 (8%)
correct.	6	*CoCo*, LLOC, MI, HVOL	0.9 (21%)	NLE, MI	0.9 (21%)
	9	*CoCo*, HCPL	0.05 (12%)	LLOC, HCPL	0.05 (11%)
	10	*CoCo*, LLOC, NLE-MI	0.15 (20%)	McCC, LLOC, MI	0.15 (20%)

B.5. Evaluations of multivariate models

Table B.18 illustrates the performance of the multivariate models by providing the MdAR and MdR of the predictions. Table B.18 confirms the results in Table 13 in that no code measure appears a consistently better predictor than other measures.

References

Ajami, S., Woodbridge, Y., Feitelson, D.G., 2019. Syntax, predicates, idioms - what really affects code complexity? Empir. Softw. Eng. 24 (1), 287–328. http://dx.doi.org/10.1007/s10664-018-9628-3.

Barón, M.M., Wyrich, M., Wagner, S., 2020a. An empirical validation of cognitive complexity as a measure of source code understandability. In: Proceedings of the 14th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement. ESEM, pp. 1–12.

Barón, M.M., Wyrich, M., Wagner, S., 2020b. An empirical validation of cognitive complexity as a measure of source code understandability – Data, code and documentation. http://dx.doi.org/10.5281/zenodo.3949828.

Beyer, D., Fararooy, A., 2010. A simple and effective measure for complex low-level dependencies. In: The 18th IEEE International Conference on Program Comprehension, ICPC 2010, Braga, Minho, Portugal, June 30-July 2, 2010. IEEE Comput. Soc., pp. 80–83. http://dx.doi.org/10.1109/ICPC.2010.49.

Börstler, J., Paech, B., 2016. The role of method chains and comments in software readability and comprehension - An experiment. IEEE Trans. Softw. Eng. 42 (9), 886–898. http://dx.doi.org/10.1109/TSE.2016.2527791.

Buse, R.P.L., Weimer, W., 2010. Learning a metric for code readability. IEEE Trans. Softw. Eng. 36 (4), 546–558. http://dx.doi.org/10.1109/TSE.2009.70.

Campbell, G.A., 2018a. Cognitive complexity - a new way of measuring understandability. URL https://www.sonarsource.com/docs/CognitiveComplexity.pdf.

Campbell, G.A., 2018b. Cognitive complexity: An overview and evaluation. In: Proceedings of the 2018 International Conference on Technical Debt. pp. 57–58.

Chen, E.T., 1978. Program complexity and programmer productivity. IEEE Trans. Softw. Eng. 4 (3), 187–194. http://dx.doi.org/10.1109/TSE.1978.231497.

Cohen, J., 2013. Statistical Power Analysis for the Behavioral Sciences. Academic Press.

Coleman, D., Ash, D., Lowther, B., Oman, P., 1994. Using metrics to evaluate software system maintainability. Computer 27 (8), 44–49.

Denaro, G., Lavazza, L., Pezze, M., 2003. An empirical evaluation of object oriented metrics in industrial setting. In: The 5th CaberNet Plenary Workshop, Porto Santo, Madeira Archipelago, Portugal.

Dolado, J.J., Harman, M., Otero, M.C., Hu, L., 2003. An empirical investigation of the influence of a type of side effects on program comprehension. IEEE Trans. Softw. Eng. 29 (7), 665–670. http://dx.doi.org/10.1109/TSE.2003.1214329.

Dunsmore, H.E., Gannon, J.D., 1979. Data referencing: An empirical investigation. Computer 12 (12), 50–59. http://dx.doi.org/10.1109/MC.1979.1658576.

Fenton, N.E., Bieman, J.M., 2014. Software Metrics: A Rigorous and Practical Approach, third ed. Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, Taylor & Francis, URL https://books.google.es/books?id=lx_OBQAAQBAJ.

Fitzsimmons, A., Love, T., 1978. A review and evaluation of software science. ACM Comput. Surv. 10 (1), 3–18.

Floyd, B., Santander, T., Weimer, W., 2017. 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, IEEE, pp. 175–186.

Fucci, D., Girardi, D., Novielli, N., Quaranta, L., Lanubile, F., 2019. A replication study on code comprehension and expertise using lightweight biometric sensors. In: 2019 IEEE/ACM 27th International Conference on Program Comprehension. ICPC, IEEE, pp. 311–322.

Glass, R.L., 2001. Frequently forgotten fundamental facts about software engineering. IEEE Softw. 18 (3), 112.

Gopstein, D., Iannacone, J., Yan, Y., DeLong, L., Zhuang, Y., Yeh, M.K., Cappos, J., 2017. Understanding misunderstandings in source code. In: Bodden, E., Schäfer, W., van Deursen, A., Zisman, A. (Eds.), Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017. ACM, pp. 129–139. http://dx.doi.org/10.1145/3106237.3106264.

Halstead, M.H., 1977. Elements of Software Science. Elsevier North-Holland.

Harrison, W.A., Magel, K.I., 1981a. A complexity measure based on nesting level. SIGPLAN Not. 16 (3), 63–74. http://dx.doi.org/10.1145/947825.947829.

Harrison, W., Magel, K., 1981b. A topological analysis of the complexity of computer programs with less than three binary branches. SIGPLAN Not. 16 (4), 51–63. http://dx.doi.org/10.1145/988131.988137.

Hedges, L.V., Olkin, I., 2014. Statistical Methods for Meta-Analysis. Academic Press.

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

Hofmeister, J.C., Siegmund, J., Holt, D.V., 2017. Shorter identifier names take longer to comprehend. In: Pinzger, M., Bavota, G., Marcus, A. (Eds.), IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017. IEEE Computer Society, pp. 217–227. http://dx.doi.org/10.1109/SANER.2017.7884623.

Howatt, J.W., Baker, A.L., 1989. Rigorous definition and analysis of program complexity measures: An example using nesting. J. Syst. Softw. 10 (2), 139–150. http://dx.doi.org/10.1016/0164-1212(89)90025-3.

Ikutani, Y., Uwano, H., 2014. Brain activity measurement during program comprehension with NIRS. In: 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing. SNPD, IEEE, pp. 1–6.

Lavazza, L., 2020. A large scale empirical evaluation of the accuracy of function points estimation methods. Int. J. Adv. Softw. 13 (3–4), 182–193.

Li, E.Y., 1987. A measure of program nesting complexity. In: 1987 AFIPS National Computer Conference. NCC, AFIPS, pp. 531–538.

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. (4), 308–320.

Minelli, R., Mocci, A., Lanza, M., 2015. I know what you did last summer-an investigation of how developers spend their time. In: 2015 IEEE 23rd International Conference on Program Comprehension. IEEE, pp. 25–35.

Morasca, S., 2009. A probability-based approach for measuring external attributes of software artifacts. In: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement. ESEM '09, Lake Buena Vista, FL, USA, October 15-16, 2009, IEEE Computer Society, Washington, DC, USA, pp. 44–55. http://dx.doi.org/10.1109/ESEM.2009.5316048.

Oman, P., Hagemeister, J., 1992. Metrics for assessing a software system's maintainability. In: Proceedings Conference on Software Maintenance 1992. IEEE Computer Society, pp. 337–338.

Osgood, C.E., Sebeok, T.A., Gardner, J.W., Carroll, J.B., Newmark, L.D., Ervin, S.M., Saporta, S., Greenberg, J.H., Walker, D.E., Jenkins, J.J., et al., 1954. Psycholinguistics: A survey of theory and research problems. J. Abnorm. Soc. Psychol. 49 (4p2), I.

Ostberg, J.-P., Wagner, S., 2014. On automatically collectable metrics for software maintainability evaluation. In: 2014 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement. IEEE, pp. 32–37.

Padilha, J., Pereira, J., Figueiredo, E., Almeida, J., Garcia, A., Sant'Anna, C., 2014. On the effectiveness of concern metrics to detect code smells: An empirical study. In: International Conference on Advanced Information Systems Engineering. Springer, pp. 656–671.

Peitek, N., Siegmund, J., Apel, S., Kästner, C., Parnin, C., Bethmann, A., Leich, T., Saake, G., Brechmann, A., 2020. A look into programmers' heads. IEEE Trans. Softw. Eng. 46 (4), 442–462. http://dx.doi.org/10.1109/TSE.2018.2863303.

Piwowarski, P., 1982. A nesting level complexity measure. ACM SIGPLAN Not. 17 (9), 44–50. http://dx.doi.org/10.1145/947955.947960.

R. core team, 2015. R: A language and environment for statistical computing.

Rosenthal, R., Cooper, H., Hedges, L., et al., 1994. Parametric measures of effect size. In: The Handbook of Research Synthesis, vol. 621, (2), New York, pp. 231–244.

Salvaneschi, G., Amann, S., Proksch, S., Mezini, M., 2014. An empirical study on program comprehension with reactive programming. In: Cheung, S., Orso, A., Storey, M.D. (Eds.), Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014. ACM, pp. 564–575. http://dx.doi.org/10.1145/2635868.2635895.

Scalabrino, S., Bavota, G., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., Oliveto, R., 2021. Automatically assessing code understandability. IEEE Trans. Softw. Eng. 47 (3), 595–613. http://dx.doi.org/10.1109/TSE.2019.2901468.

Sharafi, Z., Huang, Y., Leach, K., Weimer, W., 2021. Toward an objective measure of developers' cognitive activities. ACM Trans. Softw. Eng. Methodol. 30 (3), 1–40.

Shen, V.Y., Conte, S.D., Dunsmore, H.E., 1983. Software science revisited: A critical analysis of the theory and its empirical support. IEEE Trans. Softw. Eng. (2), 155–165.

Shepperd, M., 1988. A critique of cyclomatic complexity as a software metric. Softw. Eng. J. 3 (2), 30–36.

Shepperd, M., MacDonell, S., 2012. Evaluating prediction systems in software project estimation. Inf. Softw. Technol. 54 (8), 820–827.

Siegmund, J., Brechmann, A., Apel, S., Kästner, C., Liebig, J., Leich, T., Saake, G., 2012. Toward measuring program comprehension with functional magnetic resonance imaging. In: Tracz, W., Robillard, M.P., Bultan, T. (Eds.), 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012. ACM, p. 24. http://dx.doi.org/10.1145/2393596.2393624.

Subramanyam, R., Krishnan, M.S., 2003. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. IEEE Trans. Softw. Eng. 29 (4), 297–310.

Van Rijsbergen, C., 1979. Information retrieval: Theory and practice. In: Proceedings of the Joint IBM/University of Newcastle Upon Tyne Seminar on Data Base Systems. pp. 1–14.

Welker, K.D., Oman, P.W., Atkinson, G.G., 1997. Development and application of an automated source code maintainability index. J. Softw. Maint. 9 (3), 127–159.

Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A.E., Li, S., 2017. Measuring program comprehension: A large-scale field study with professionals. IEEE Trans. Softw. Eng. 44 (10), 951–976.

Luigi Lavazza is associate professor at the University of Insubria at Varese, Italy. Formerly he was assistant professor at Politecnico di Milano, Italy. Since 1990 he cooperates with the Software Engineering group at CEFRIEL, where he acts

as a scientific consultant in digital innovation projects. His research interests include: Empirical software engineering, software metrics and software quality evaluation; Software project management and effort estimation; Software process modeling, measurement and improvement; Open Source Software. He was involved in several international research projects, and he also served as reviewer of EU projects. He is co-author of over 180 scientific articles, published in international journals, in the proceedings of international conferences or in books. He has served on the PC of a several of international Software Engineering conferences, and in the editorial board of international journals.

Abedallah Zaid Abualkishik is Associate Professor of Software Engineering. He is passionate about the managerial process of software measurement and development, coding themes, Big Data, Blockchain and Data Science. His research interests include software functional size measurement, software functional size, measures conversion, cost estimation, empirical software engineering, code understandability, database, Big Data and Data Science. Dr. Abedallah started his earlier career as an assistant professor at KIC in Abu Dhabi, UAE. He worked as an adjunct assistant professor at Abu Dhabi University, UAE. Currently, he is an Associate Professor and the department chair of the Computer Science program at the American University in the Emirates, Dubai, UAE. Dr. Abedallah has published several highly reputable refereed papers in high impact factor journals and international conferences. He is serving the scientific community as a regular reviewer for several journals. In addition, he is working as a consultant for a regional software development company to prepare accurate estimation for project deliverables. Furthermore, he is a certified IBM Big Data Engineer, certified IBM Big Data developer, certified IBM Blockchain developer, certified IBM AI developer, and certified IBM data science developer.

Geng Liu is a lecturer at Hangzhou Dianzi University, China. He received the Ph.D. degree in Computer Science from University of Insubria (Italy) in 2014. His research interests include, among others, Software Metrics, Software Quality, Requirements Modeling and Requirement-driven Design.

Sandro Morasca is a Professor of Computer Science at the Dipartimento di Scienze Teoriche e Applicate of the Università degli Studi dell'Insubria in Como and Varese, Italy. He was an Associate and Assistant Professor at the Politecnico di Milano in Milano and Como, Italy, and a Faculty Research Assistant and later a Visiting Scientist at the Department of Computer Science of the University of Maryland at College Park. Sandro Morasca has been actively carrying out research in Empirical Software Engineering, Software Quality, Machine Learning, Software Verification, Open Source Software, Web Services, and Specification of Concurrent and Real-time Software Systems, and has published over 30 journal papers and over 90 conference papers. Sandro Morasca has been involved in several national and international projects and has served on the Program Committees and Editorial Boards of international Software Engineering conferences and journals.