Empirical Software Engineering (2023) 28:155
https://doi.org/10.1007/s10664-023-10396-7

# An empirical study on software understandability and its dependence on code characteristics

**Luigi Lavazza[1] · Sandro Morasca[1] · Marco Gatto[1]**

Accepted: 18 September 2023
© The Author(s) 2023

## Abstract

**Context** Insufficient code understandability makes software difficult to inspect and maintain and is a primary cause of software development cost. Several source code measures may be used to identify difficult-to-understand code, including well-known ones such as Lines of Code and McCabe's Cyclomatic Complexity, and novel ones, such as Cognitive Complexity.
**Objective** We investigate whether and to what extent source code measures, individually or together, are correlated with code understandability.
**Method** We carried out an empirical study with students who were asked to carry out realistic maintenance tasks on methods from real-life Open Source Software projects. We collected several data items, including the time needed to correctly complete the maintenance tasks, which we used to quantify method understandability. We investigated the presence of correlations between the collected code measures and code understandability by using several Machine Learning techniques.
**Results** We obtained models of code understandability using one or two code measures. However, the obtained models are not very accurate, the average prediction error being around 30%.
**Conclusions** Based on our empirical study, it does not appear possible to build an understandability model based on structural code measures alone. Specifically, even the newly introduced Cognitive Complexity measure does not seem able to fulfill the promise of providing substantial improvements over existing measures, at least as far as code understandability prediction is concerned. It seems that, to obtain models of code understandability of acceptable accuracy, process measures should be used, possibly together with new source code measures that are better related to code understandability.

Communicated by: Simone Scalabrino, Rocco Oliveto, Felipe Ebert, Fernanda Madeiral, Fernando Castor

This article belongs to the Topical Collection: *Special Issue on Code Legibility, Readability, and Understandability*.

This work was partly supported by the "Fondo di ricerca d'Ateneo" funded by the Università degli Studi dell'Insubria.

✉ Luigi Lavazza
luigi.lavazza@uninsubria.it

Extended author information available on the last page of the article

Published online: 15 November 2023

**Keywords** Software understandability · Cognitive complexity · Software code measures · Complexity measures · Static code measures

## 1 Introduction

Software professionals spend a large amount of time and effort understanding software code (Minelli 2015; Xia et al. 2017). It would be very useful to be able to predict in advance with a sufficient degree of confidence which sections of software code are difficult to understand, so as to take adequate action. For instance, hard-to-understand software code could be revised to improve its clarity, to make following maintenance activities easier and less time- and effort-consuming. In addition, proactive rules could be established to avoid writing unreadable code in the first place.

Prediction of various software qualities, e.g., software fault-proneness, has been often carried out by building models that relate them with one or more code measures. Likewise, accurately modeling the relationship between code understandability and some code measures would be quite useful for software development. A large number of code measures exist. Some of them, e.g., the number of Lines of Code, McCabe's Cyclomatic Complexity (which we denote as *McCC*) (McCabe 1976), various Maintainability Indices (Heitlager et al. 2007; Oman and Hagemeister 1992), and Halstead measures (Halstead 1977) have been used to this end in the past. None of them has however led to the building of very accurate prediction models that could be used in practice to predict and control code understandability. Other measures, notably Cognitive Complexity (Campbell 2018), have recently been introduced with the goal of taking into account understandability-related aspects that previous measures do not capture. However, no empirical evidence has so far been provided to support their usefulness.

We carried out an empirical study with the goal of investigating whether various source code measures could be useful to build accurate understandability prediction models and whether novel measures such as Cognitive Complexity could be useful to this end.

Specifically, our empirical study addresses the following Research Questions.

**RQ1** Is it possible to define predictive models for code understandability by using source code measures?

**RQ2** Which source code measures are best at predicting code understandability?

**RQ3** Is Cognitive Complexity better than other measures as a predictor of code understandability?

**RQ4** How much can source code measures be trusted as understandability predictors? In other words, are predictions based on code measures accurate enough? Do we need to consider additional factors, other than code characteristics, that affect understandability?

The answers to the Research Questions help provide practitioners with indications on which methods are more difficult to understand, hence are more likely to cause problems. This will lead to reduced maintenance costs.

Our empirical study involved three Master's degree students who carried out realistic maintenance tasks on 32 methods from two real-life Open Source Software applications. We measured understandability as the time needed to correctly complete the maintenance tasks. We also collected several source code measures and investigated whether and to what extent they are correlated to the understandability measure. To this end, we used several Machine Learning techniques.

The models we obtained in our empirical study show that code understandability is correlated with structural characteristics of code. Nonetheless, the obtained models are not

accurate; also the usage of the recently introduced Cognitive Complexity measure does not seem to help. Thus, code understandability depends only partially on code characteristics, as quantified by the measures chosen. This result could be expected, since code understandability depends on the code but also, to a great extent, on *who* has to understand the code. This suggests that future research activities should take the human factor in due account.

The remainder of the 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 Cognitive Complexity, 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 accounts for related work. Section 9 illustrates the conclusions and outlines future work.

## 2 Source Code Understandability

A large part of the costs incurred during the software development process, even the majority (60% on average, according to Glass 2001), is due to software maintenance and evolution activities. Maintaining software implies being able to fully understand code that was written by the maintainers themselves or by other developers. The lack of familiarity of maintainers with the software code they deal with is one of the main causes of the large amount of effort that maintainers spend understanding code (Minelli 2015). Given their industrial importance, software maintainability and understandability have been the subject of several empirical studies.

Maintainability and understandability belong to the category of "external" software attributes (Fenton and Bieman 2014). 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 and understood. This is apparent because the amount of effort needed by developers in maintaining and understanding code that they wrote is certainly lower than the amount of effort that other maintainers would spend in maintaining or understanding the very same code.

The research documented in this paper focuses on understandability. Like many other software attributes, understandability has different aspects and may therefore be measured in several different ways, based on the goals and constraints of software development and Empirical Software Engineering research. We concisely summarize how understandability has been measured in previous studies.

- *Time*. An often used understandability measure is the time taken to carry out some comprehension task on some software code (Ajami et al. 2019; Börstler and Paech 2016; Dolado et al. 2003; Hofmeister et al. 2017; Peitek et al. 2020; Salvaneschi et al. 2014; Scalabrino et al. 2021; Siegmund et al. 2012).
- *Correctness*. Correctness is usually defined as the degree of success in performing one or more specified maintenance tasks that require understanding a specific portion of software code (Börstler and Paech 2016; Dolado et al. 2003; Salvaneschi et al. 2014; Scalabrino et al. 2021; Siegmund et al. 2012).
- Subjective *rating*. This is the subjective perception of how well maintainers believe that they understood the code they are asked to maintain, usually on an ordinal scale (Börstler and Paech 2016; Buse and Weimer 2010; Scalabrino et al. 2021).

– *Physiological* measures. Several studies (Floyd et al. 2017; Fucci et al. 2019; Ikutani and Uwano 2014; Peitek et al. 2020; 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.

In empirical studies like ours, one or more of these measures were taken as the dependent variable of models whose independent variables are source code measures and possibly others.

We here use a time-related measure of understandability, specifically, the time needed to correctly carry out a maintenance task on a software method. This is therefore the dependent variable of the models we build in our empirical study. More details about the measurement of this dependent variable are in Section 4, which describes the empirical study.

## 3 Source Code 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 considered source code measures that have been present in the literature and in practical use for several years. We concisely describe them in Section 3.1. In addition, we considered Cognitive Complexity, a novel measure that was introduced with the purpose of overcoming the pitfalls of existing measures (Campbell 2018). Throughout the paper we name this measure *CoCo*, to avoid confusion with the actual cognitive complexity, i.e., what *CoCo* is supposed to evaluate. We describe *CoCo* more extensively in Section 3.2.

### 3.1 Traditional Code Measures

Here are the measures from the literature that we took into account.

**Logical Lines of Code** Software size, measured via the number of lines of code, is the first code characteristic to be quantified. *LOC* (or the logical *LOC*, i.e., *LLOC*) are so widely used that performing a study that involves code measures without considering *LOC* (or *LLOC*) is almost inconceivable.

**McCabe's Complexity** *McCC* was originally proposed to identify software modules that are difficult to test or maintain, based on the control flow of a function or method. *McCC* has been used extensively as an indicator of difficult understandability and maintainability.

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

**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 occurrences of operators, $N_2$ is total number of occurrences of operands; $\eta = \eta_1 + \eta_2$ is the "program vocabulary," $\eta_1$ is the number of distinct operators and $\eta_2$ is the number of distinct operands. According to Fitzsimmons and Love, "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." (Fitzsimmons and Love 1978)

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

$$HCPL = \eta_1 * \log_2(\eta_1) + \eta_2 * \log_2(\eta_2) \tag{2}$$

**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:

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

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 accuracy than those measures.

We used SourceMeter[1] to collect these measures, because it is a fairly consolidated and robust tool and it is efficient and well documented.

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 for completeness and as a sort of benchmark for a more thorough evaluation of *CoCo*, which we describe next.

### 3.2 The "Cognitive Complexity" Measure

In 2018, SonarSource introduced "Cognitive Complexity" (Campbell 2018) as a new measure for the understandability of a given piece of code. *CoCo* takes into account several aspects of code. Like McCabe's complexity, it takes into account decision points (conditional statements, loops, switch statements, etc.), but, unlike McCabe's complexity, it 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.

1 https://www.sourcemeter.com/

```
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*. This inadequacy of *McCC* is one of the main reasons for the introduction of *CoCo* (Campbell 2018). One of the goals of our empirical study is to check whether this difference between *McCC* and *CoCo* can help successfully identify the code that is hard to understand and, if so, to what extent.

*CoCo* also accounts for the nesting level (which is also measured by *NLE*). Hence, it is reasonable to expect that *CoCo* is able to identify hard-to-understand code more effectively than *McCC* or *NLE* alone. 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 hard-to-understand code. Our empirical study was designed to gather some evidence.

The structure of Boolean predicates is also taken into account by *CoCo*. 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 sub-sequences 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 2018).

## 4 The Empirical Study

Ideally, the best way of evaluating code understandability is by observing professionals at work. However, this is hardly ever possible, thus we resorted to an empirical study involving Master's students.

### 4.1 Objectives and Conditions

We pursued the following two objectives.

– Analyze code understandability in realistic conditions. To this end, we analyzed the understandability of pieces of code of realistic size and complexity. We also devised tasks that resemble as much as possible part of the actual work carried out by professional programmers. As a result, the unit of code to be understood is the method.

– Use a process to build models that is feasible for practitioners. Thus, we used techniques that are available in a "packaged" way that makes them usable without very sophisticated knowledge of data analysis techniques.

We must consider that understandability is an external property, i.e., it does not depend exclusively on code properties, but also on additional properties and conditions, with special reference to who has to understand the code. In this sense, one does not measure code understandability, but how quickly, correctly, etc., someone understood a piece of code. To make our results as independent as possible from the context and the participants, we enrolled a set of participants as homogeneous as possible in the empirical study.

As stated above, our study involves analyzing code understandability in conditions that can be found in real code maintenance activities. Nonetheless, typical code maintenance processes can as well be carried out in conditions that are not covered by our study: for instance, maintainers may know the code and may very well talk to each other. This should not be regarded as a limitation of our study: in fact, the aim of this study is to collect quantitative data that represent code understandability, not to collect data that can be representative of all the possible maintenance processes.

### 4.2 Organization and Execution

We tried to re-create and simulate real-world scenarios that developers may encounter in their activities.

One of the main problems in an experiment dealing with code comprehension is how to quantitatively characterize software understandability. Given also the kind of tasks in which participants were involved, we aimed to collect two main measures:

– overall time required to solve a comprehension task;
– correctness of the proposed solution.

In the experiment, we did not enforce time limits for each task. As a result, all participants were able to provide correct solutions, eventually. Thus, we measured code understandability via the time taken by each participant to produce a correct solution.

To ensure the homogeneity of participants, we involved Master's students in Computer Science, all having similar levels of knowledge of the coding language and similar levels of programming experience. The coding language used is Java, because it is the language most diffusely used in courses. In practice, the proficiency in Java programming of the involved students may be deemed similar to that of junior professionals (Carver et al. 2010).

Tasks require that the participants carry out some defect removal operations. Hence, tasks involve faulty methods, which are of two types:

1. the method does not call other methods, thus the comprehension activity scope is limited to the considered method;
2. the method calls other methods, thus the comprehension activity scope includes the considered method and the methods it uses.

In our empirical study, it is possible that a faulty method $m_1$ calls a method $m_2$ that is also faulty. In these cases, a participant was always given a list of methods to be corrected involving both $m_1$ and $m_2$.

To make the experiment feasible, a way to evaluate methods' correctness was needed. To this end, every method is equipped with a set of unit tests that assess the correct behavior of a

method. So, correctness could be quickly evaluated by the participants during the empirical study as well as by the supervisor, who needed to evaluate the correctness of the modified code supplied by participants.

Participants were supplied with

- the code of an open-source project;
- a list of defective methods belonging to the supplied code;
- a set of unit tests for each of the defective methods.

For each method in the list, participants had to locate the fault in the method, devise a way to correct the faulty method, perform the correction, and test the modified code by running the available test cases.

The nature of the faults was such that understanding the code to locate the fault was by far the most challenging and time-consuming activity, since the required corrections were trivial and the testing just required running the corresponding unit test case. Therefore, coding and testing took negligible time. So, although the measures we collected involve coding and testing time as well, we can regard them as proper measures of code understandability.

As a final remark, we can note that according to a systematic literature review (Oliveira et al. 2020) several studies addressing code readability and legibility considered time and correctness of tasks involving finding and fixing bugs.

#### 4.2.1 Choice of the Source Code Used in the Empirical Study

We used Open Source code because that is the easiest way to get access to code. Within the many available open-source projects that use the Java language, we needed to select projects satisfying the following conditions.

- Methods should be equipped with test cases.
- Applications should not be too large, because participants should not get lost in a complex project structure.
- Code understanding should not require specific application domain knowledge that the participants do not master. Accordingly, we restricted the choice to applications in the information technology field.
- Finally, we needed projects whose methods were sufficiently varied with respect to the considered code metrics. That is, we needed methods having different size, complexity, *CoCo*, etc.

Two applications concerning the processing of JSON files were finally selected: JSON-Java (GitHub - stleary/JSON-java 2022) and Jsoniter (GitHub 2022).

The specific methods to be used in the experiment were chosen based on the following criteria:

- They should be equipped with test cases. In general, not all methods have test cases, even in projects where unit tests are used.
- They were associated with some solved issues. This allowed us to identify the buggy versions of the methods to be proposed to participants. Thus, we also made sure that the tasks assigned to participants were realistic ones.
- They should form sets sufficiently varied with respect to the considered code metrics (size, complexity, *CoCo*, etc.).

Based on these criteria we selected 16 methods from each project, for a total of 32 methods.

#### 4.2.2 Empirical Study Execution

Three students participated in the empirical study. They were recruited among the students attending the Master's Degree in Computer Science program at the Università degli Studi dell'Insubria. They had similar expertise concerning the programming language and the application domain, acquired through university courses. All of them had also a few months of work experience acquired via the industrial training required for the Bachelor Degree.

The empirical study was carried out in two sessions, lasting four hours each, in different days, to avoid fatigue effects. In each session, each participant had to perform the corrective maintenance of eight methods (four from Jason-java and four from Jsoniter). Half of the methods were assigned to multiple participants. This was done to be able to evaluate participants. In fact, participants' ability is a potentially confounding factor that has to be evaluated and taken into account.

Suitable working conditions were created. Participants used the Eclipse IDE on their own machine, and they could take breaks, whose duration was not counted in task execution time.

Participants were properly instructed not to communicate with each other, and they were also informed that they were not being evaluated in any way via the empirical study. To make the environment as friendly as possible, sessions were supervised by another Master's student.

The tracking of the times took place through the push system on the repository, reporting both the provided solution and the completion time of each task.

### 4.3 Collected Data

For each method involved in a task, the following data were collected via the experiment:

- Id of the participant that performed the maintenance task;
- Name of the method;
- Class of the method;
- Time in minutes taken to provide the solution;
- Correctness of the solution (Boolean).

As already mentioned, all methods were successfully corrected by all participants, i.e., correctness was true for all supplied methods.

In addition, every method was measured via SourceMeter (Source 2022). The measures mentioned in Section 3 were collected. Table 1 contains the descriptive statistics for the data we collected.

**Table 1** Descriptive statistics of the Java methods used in the empirical study

| | Time | CoCo | HVOL | HCPL | McCC | LLOC | NLE | MI |
|---|---|---|---|---|---|---|---|---|
| mean | 28.1 | 16.5 | 936.1 | 271.2 | 10.7 | 33.6 | 2.8 | 78.2 |
| st. dev. | 11.8 | 10.9 | 418.4 | 97.0 | 6.5 | 14.8 | 1.5 | 11.2 |
| median | 26.5 | 12.0 | 838.7 | 254.1 | 10.0 | 31.0 | 3.0 | 77.0 |
| min | 9 | 2 | 244 | 104 | 1 | 10 | 0 | 59 |
| max | 77 | 43 | 1956 | 522 | 28 | 68 | 7 | 105 |

# 5 Data Analysis and Results

## 5.1 Evaluation of Participants

Code understanding depends on code as well as on who has to understand the code. It is therefore important to evaluate the differences in code understanding ability of the participants. To this end, we proceeded as follows: we identified the set of methods assigned to two or more participants, then we collected the time employed by the involved participants to complete the tasks concerning these common methods. These data are given in Table 2, where P1, P2 and P3 indicate the three participants in the study. To highlight the differences, for each common task we computed the mean completion time $\bar{t}$ and the relative difference to $\bar{t}$ mean (RDtM) for each participant (for instance, in the first row, concerning method `fillCacheUntil`, the mean time is $\frac{29+26+32}{3} = 29$ and the RDtM for participant 2 is $\frac{26-29}{29} = -10.3\%$.

It can be seen that different participants obtained similar results for common methods, except for method `nextValue` and, to a lesser extent, for method `objectToBigInteger`. We also evaluated the global performances of participants, computed as the mean times taken by participants to complete the assigned tasks. They are shown in the bottom line of Table 2 and they appear to be very similar.

It is also worth noting that no participant was consistently better or worse than others: specifically, the RDtM column of each participant includes both positive and negative values. This is what we expect from developer groups in real organizations: a developer performs better than colleagues in some tasks and worse in others.

**Table 2** Participants' common task completion times

| | Time | | | RDtM | | |
|---|---|---|---|---|---|---|
| Method | P1 | P2 | P3 | P1 | P2 | P3 |
| Any fillCacheUntil(int target) | 29 | 26 | 32 | 0.0% | -10.3% | 10.3% |
| BigInteger objectToBigInteger(...) | 35 | 22 | NA | 22.8% | -22.8% | |
| Boolean equals(Object o) | NA | 19 | 17 | | 5.6% | -5.6% |
| Decoder createDecoder(...) | 24 | 25 | 32 | -11.1% | -7.4% | 18.5% |
| Int findStringEnd(JsonIterator iter) | 22 | NA | 29 | -13.7% | | 13.7% |
| Int parse(JsonIterator iter) | 18 | 22 | NA | -10.0% | 10.0% | |
| JSONArray(JSONTokener x) | NA | 28 | 25 | | 5.7% | -5.7% |
| Object nextMeta() | 27 | NA | 38 | -16.9% | | 16.9% |
| Object nextValue() | NA | 77 | 30 | | 43.9% | -43.9% |
| Object read() | 25 | 32 | NA | -12.3% | 12.3% | |
| String toString(JSONArray ja) | 23 | 27 | NA | -8.0% | 8.0% | |
| String unescape(String string) | 19 | 20 | 24 | -9.5% | -4.8% | 14.3% |
| Void enableDecoders() | 37 | NA | 33 | 5.7% | | -5.7% |
| Void populateMap(Object bean) | 41 | NA | 33 | 10.8% | | -10.8% |
| Void skipFixedBytes(...) | NA | 16 | 16 | | 0.0% | 0.0% |
| Writer write(Writer writer, ...) | 26 | 32 | 38 | -18.8% | 0.0% | 18.8% |
| Mean time for all methods | 27.2 | 28.8 | 28.9 | -4.0% | 1.9% | 2.2% |

### 5.2 Analysis Methods

At first, we tried building models via Ordinary Least Squares (OLS) regression, both linear and after log-log transformation, given that the distribution of data is not normal. However, the resulting models were quite inaccurate. Therefore, we tried applying more sophisticated Machine Learning techniques.

We built models using Support Vector Regression (SVR), Random Forests (RF) and Neural Networks (NN) approaches. The analysis was carried out using the R programming language and environment (R core team 2015). Specifically, we used the `e1071`, `nnet`, and `randomForest` libraries.

The models obtained with different techniques provided generally concordant indications. NN models provided slightly more accurate predictions then the other models, hence we report only the results from NN models.

To build models, a fundamental step was the configuration of ML model with proper parameters (i.e., the so-called hyperparameters of the model), in order to find the best configuration for our dataset. To this end, we exploited the `tuning` function of the `e1071` library, which has a wrapper for many ML methods available in R. The `tuning` function was designed to find the best set of parameters for the data in a ranged or full parameter space for each parameter. 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. Hyperparameters values and ranges were set after a preliminary test phase, where subsequently simplified parameter spaces were used and compared against time performance, accuracy intervals, and "best practice" hyperparameters tuning.[2] The final model-specific parameters for NN models were set as follows.

The `nnet` package models a single hidden layer neural network; `linout` (for linear output instead of logistic output) = `true`; `rang` (initial random weights interval [-`rang`, `rang`]) = 0.1. Best parameter space considered: `size` (number of units in the hidden layer) = 2, `decay` (a factor by which the minimization of the loss function procedure is affected, in that it "regularizes" weights value at each step) = 0.

Models were built and evaluated via 10-times 10-fold cross validation, i.e., the dataset was split randomly in ten subsets, and each subset was used as a test set to evaluate the model built on the basis of the other data. The procedure was repeated 10 times to average out the effects of random splitting.

Prediction accuracy 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 $i^{th}$ prediction $\hat{y}_i$ is $y_i - \hat{y}_i$, where $y_i$ is the $i^{th}$ observation (i.e., the actual value of the considered understandability factor). MAR is then computed as the mean of absolute residuals, as follows:

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

MAR is useful to compare the accuracy of different models, but does not provide a clear perception of how good the model is. To this end, we provide also a normalized version of MAR, obtained by dividing MAR by the mean actual value of the considered aspect, i.e., the time taken to carry out tasks. Specifically, we proceed as follows: given a set of observations $Y$,

[2] The e1071 package uses the Grid Search meta-heuristic for parameters tuning, given a parameter space in input.

- The residual of the $i^{th}$ prediction $\hat{y}_i$ is $y_i - \hat{y}_i$, where $y_i$ is the $i^{th}$ 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:

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

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).

MdAR is the median of absolute errors. MdR is computed as MdAR divided by the median of understanding times.

Accuracy metrics MAR, MdAR, MR, and MdR are loss functions (or penalty functions), so they are smaller in more accurate models. No consensus thresholds exist for them to separate models that are accurate enough from those that are not accurate enough. This assessment is therefore carried out by software practitioners based on their goals. At any rate, the used metrics always make it possible to rank different models according to their accuracy.

## 5.3 Results

In this section, we present the results obtained with NN models. As explained above, other techniques provided similar models, only slightly less accurate.

### 5.3.1 Models Using One Metric as Independent Variable

We obtained models of code understanding time using each of the considered code metrics, for which we computed MAR, MdAR, MR, and MdR, as reported in Table 3, where the best result for each column is in bold (hence, *McCC* has the best values for all four accuracy metrics).

Figure 1 shows the boxplots of the absolute errors from models that use one metric as the independent variable. Specifically, the boxplots on the left-hand side illustrate the complete sets of absolute errors, while the boxplots on the right-hand side do not show outliers, for the sake of readability. The orange diamonds represent the mean value, i.e., MAR.

Similarly, Fig. 2 shows the boxplots of the absolute relative errors from models that use one metric as the independent variable.

**Table 3** Accuracy of models using just one independent variable

| | MAR | MdAR | MR | MdR |
|---|---|---|---|---|
| CoCo | 8.8 | 6.6 | 31.3% | 25.0% |
| HVOL | 9.0 | 7.1 | 31.9% | 26.8% |
| HCPL | 8.0 | 6.4 | 28.5% | 24.2% |
| McCC | **7.6** | **5.1** | **26.9%** | **19.4%** |
| LLOC | 8.7 | 6.5 | 31.0% | 24.5% |
| NLE | 8.6 | 6.2 | 30.7% | 23.4% |
| MI | 8.7 | 6.2 | 30.8% | 23.5% |

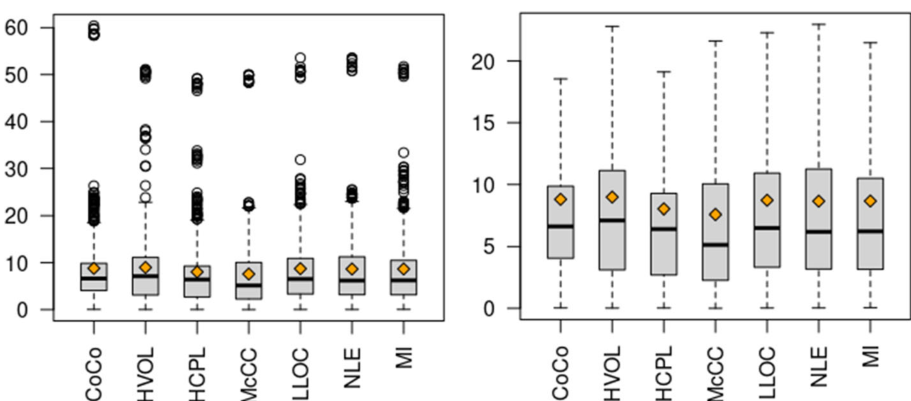


**Fig. 1** Boxplots of absolute errors from models using one independent variable (to improve readability, outliers are not shown in the right-hand side figure)

The data and a visual analysis of the boxplots show that the models have similar accuracy. In fact, Table 3 shows small ranges for the four accuracy metrics: MAR $\in$ [7.6, 9.0], MdAR $\in$ [5.1, 7.1], MR $\in$ [26.9%, 31.9%], and MdR $\in$ [19.4%, 26.8%].

The results do not seem to be very satisfactory. For instance, even the best MR (i.e., 26.9%, obtained with the *McCC*-based model) shows that the average absolute error is more than one-fourth the average time needed to complete a task. This is probably hardly acceptable.

At any rate, even though the ranges of the accuracy metrics are quite small, we proceeded to quantitatively assess to what extent a code metric is a better predictor of understandability than another metric. To this end, we computed the effect size of absolute errors, using Vargha and Delaney's $A$ (Vargha and Delaney 2000).

The Vargha and Delaney's $A$ statistic is a non-parametric effect size measure that compares the results obtained by applying two data analysis algorithms $alg_1$ and $alg_2$. Given an accuracy measure $AM$, $A$ measures the probability that the value $am_1$ of the accuracy

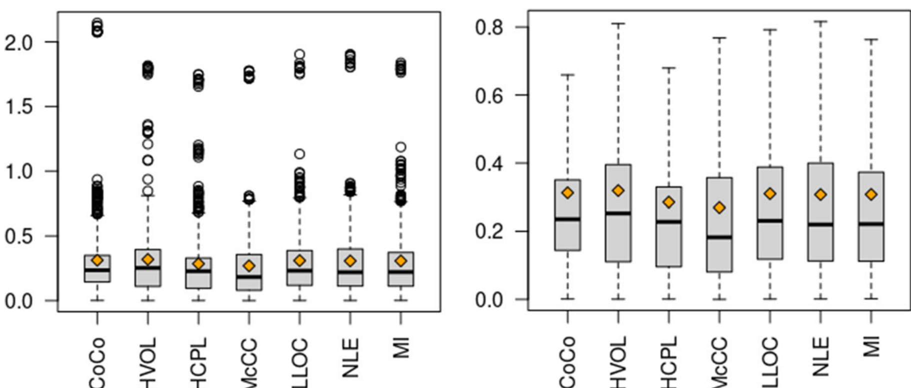


**Fig. 2** Boxplots of absolute relative errors from models using one independent variable (to improve readability, outliers are not shown in the right-hand side figure)

measure obtained by running $alg_1$ is higher than the value $am_2$ obtained by running $alg_2$. If the two algorithms have equivalent accuracy as quantified by *AM*, then $A = 0.5$. If instead, say, $A = 0.7$, we would likely obtain higher results 70% of the times with $alg_1$. *A* is defined as follows Vargha and Delaney (2000):

$$A = \frac{1}{n_2}\left(\frac{R1}{n_1} - \frac{n_1 + 1}{2}\right) \tag{4}$$

where *R1* is the rank sum of the results obtained with $alg_1$ after combining the results obtained with the two algorithms in a single set. For example (as in Arcuri and Briand 2014), assume that the values obtained with $alg_1$ are $AMValues_1 = \{21, 12, 9\}$ and those obtained with $alg_2$ are $AMValues_2 = \{4, 17, 6\}$. The values in set $AMValues_1$ have ranks 6, 4, 3, whose sum is 13. In (4), $n_1$ is the number of values obtained with $alg_1$, whereas $n_2$ is the number of values obtained with $alg_2$. Hence, in our example, $A = \frac{1}{3}(\frac{13}{3} - \frac{4}{2}) \simeq 0.78$.

The results of applying Vargha and Delaney's *A* statistic to the absolute errors of our models' predictions are given in Table 4.

Table 4 shows that the effect size is mostly negligible. We have a small effect size in only a few cases, the largest of which involves the comparison between *CoCo* and *McCC* ($A = 0.57$), which seems to indicate that *McCC* is a better predictor of understandability time than *CoCo*, though only marginally so.

Since there is no code metric that appears to be a much better predictor of understandability time than any other code metric, we proceeded to investigate the accuracy of models with multiple independent variables.

### 5.3.2 Models Using Multiple Metrics as Independent Variables

When building models with multiple code metrics as independent variables,

1. We limited the number of independent variables to four, because using more variables could imply overfitting, given the size of the dataset. This does not appear as a relevant limitation, since models using more variables do not provide more accurate predictions than those using fewer variables.
2. We did not use HVOL, because it is strongly correlated to HCPL, as shown in Formula (3).
3. Similarly, we did not use MI, since it is strongly correlated to other measures, being defined as a combination of *HVOL*, *McCC*, and *LLOC*.

**Table 4** Effect size for absolute residuals, according to Vargha and Delaney's *A*

| | CoCo | HVOL | HCPL | McCC | LLOC | NLE | MI |
|---|---|---|---|---|---|---|---|
| CoCo | — | 0.49 | 0.54 | 0.57 | 0.51 | 0.51 | 0.53 |
| HVOL | 0.51 | — | 0.55 | 0.57 | 0.51 | 0.51 | 0.52 |
| HCPL | 0.46 | 0.45 | — | 0.53 | 0.47 | 0.47 | 0.48 |
| McCC | 0.43 | 0.43 | 0.47 | — | 0.44 | 0.44 | 0.45 |
| LLOC | 0.49 | 0.49 | 0.53 | 0.56 | — | 0.50 | 0.51 |
| NLE | 0.49 | 0.49 | 0.53 | 0.56 | 0.50 | — | 0.51 |
| MI | 0.47 | 0.48 | 0.52 | 0.55 | 0.49 | 0.49 | — |

Springer

We obtained models for all the possible combinations of measures. The accuracy of the obtained models is summarized in Table 5.

The boxplots of relative absolute errors are given in Figs. 3 and 4. In Fig. 4, the boxplots are colored differently depending on the number of measures used as independent variables: light blue indicates two measures, light yellow three measures, and pink four measures.

The comparison of the results above with those for models with only one variable (reported in Section 5.3.1) shows that using multiple variable to predict code understandability does not appear to improve accuracy.

The comparison of Tables 3 and 5 shows that adding further code metrics does not improve the accuracy of models. Table 5 shows the following ranges for the four accuracy metrics: MAR $\in$ [7.6, 9.0] (with the exception of the model with independent variables *CoCo* and *NLE*, which has MAR=10.7); MdAR $\in$ [5.2, 7.2]; MR $\in$ [28.9%, 34.1%] (with the exception of the model with independent variables *CoCo* and *NLE*, which has MR=37.9%); and MdR $\in$ [19.6%, 27.1%]. There seems to be an improvement for MdAR, while all other accuracy metrics seem to indicate a marginal worsening of the accuracy.

**Table 5** Accuracy of models using multiple independent variables

| Metrics | MAR | MdAR | MR | MdR |
|---|---|---|---|---|
| CoCo,HCPL | 8.5 | 6.5 | 30.4% | 24.5% |
| CoCo,LLOC | 9.1 | 6.9 | 32.2% | 25.9% |
| CoCo,McCC | 8.6 | 6.0 | 30.4% | 22.8% |
| CoCo,NLE | 10.7 | 6.0 | 37.9% | 22.5% |
| HCPL,LLOC | **8.1** | 5.9 | **28.9%** | 22.4% |
| HCPL,McCC | 8.2 | **5.2** | 29.0% | **19.6%** |
| HCPL,NLE | 8.7 | 6.4 | 31.0% | 24.3% |
| LLOC,NLE | 8.3 | 5.9 | 29.5% | 22.1% |
| McCC,LLOC | 8.7 | 6.0 | 30.7% | 22.7% |
| McCC,NLE | 8.3 | 6.5 | 29.6% | 24.4% |
| CoCo,HCPL,McCC | 8.9 | 6.4 | 31.7% | 24.1% |
| CoCo,HCPL,LLOC | 8.6 | 6.2 | 30.4% | 23.2% |
| CoCo,HCPL,NLE | 8.8 | 5.9 | 31.4% | 22.4% |
| CoCo,LLOC,NLE | 9.5 | 6.2 | 33.9% | 23.4% |
| CoCo,McCC,LLOC | 9.1 | 7.0 | 32.3% | 26.3% |
| CoCo,McCC,NLE | 9.6 | 5.6 | 34.1% | 21.2% |
| HCPL,LLOC,NLE | 8.6 | 6.1 | 30.7% | 23.0% |
| HCPL,McCC,LLOC | 8.4 | 5.5 | 30.0% | 20.9% |
| HCPL,McCC,NLE | 8.5 | 6.0 | 30.2% | 22.8% |
| McCC,LLOC,NLE | 8.3 | 5.9 | 29.3% | 22.2% |
| CoCo,HCPL,LLOC,NLE | 9.1 | 6.3 | 32.3% | 23.6% |
| CoCo,HCPL,McCC,LLOC | 9.5 | 7.2 | 33.9% | 27.1% |
| CoCo,HCPL,McCC,NLE | 9.3 | 6.3 | 33.2% | 23.8% |
| CoCo,McCC,LLOC,NLE | 8.9 | 5.8 | 31.7% | 21.8% |
| HCPL,McCC,LLOC,NLE | 8.5 | 6.0 | 30.3% | 22.6% |

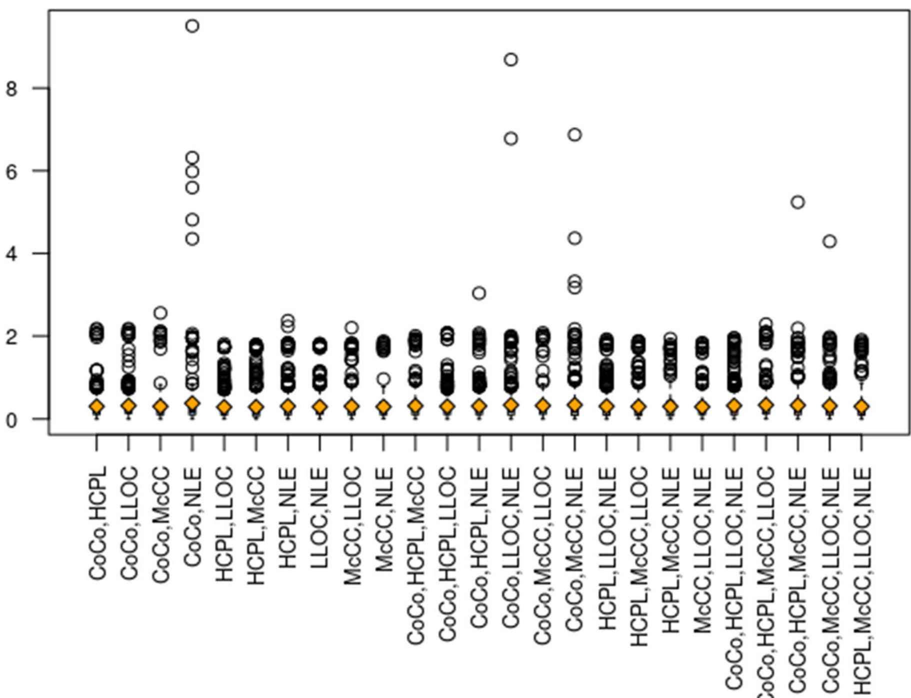


**Fig. 3** Boxplots of absolute relative errors from models using multiple independent variables

## 6 Synthesis of Findings

In Section 6.1 below we provide answers to the research questions, based on the collected data.

To verify hypotheses and conclusions, we interviewed the participants in the study, as well as experienced industrial developers. The outcome of the interviews is summarized in Section 6.2.

### 6.1 Answers to Research Questions

**RQ1** Is it possible to define predictive models for code understandability by using source code measures? We were able to find models using different ML techniques. All of the considered code metrics support code understandability models.

**RQ2** Which source code measures are best at predicting code understandability?
Table 3 and Figs. 1 and 2 show that McCabe's complexity seems to be the best predictor of code understandability, but by a very small margin. The evaluation of the effect size shows that the advantage of using *McCC* over other metrics is small, at best. In fact, all models based on a single metric have accuracy in the [26.9%, 31.9%] range, when considering absolute estimation errors.

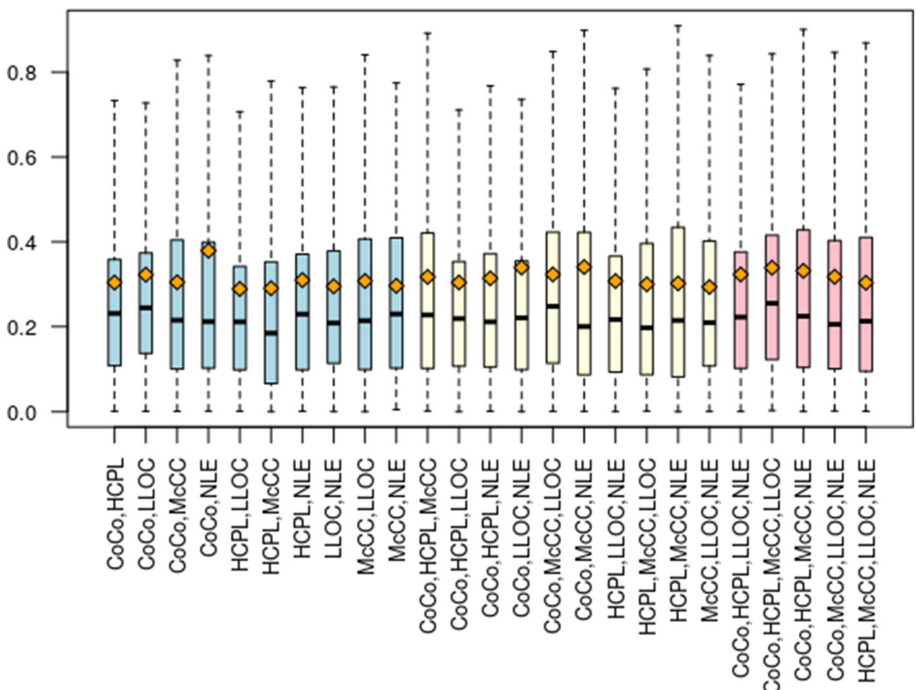


**Fig. 4** Boxplots of absolute relative errors from models using multiple independent variables (outliers not shown)

**RQ3** Is Cognitive Complexity better than other measures as a predictor of code understandability?
Based on our results, it seems that *CoCo* does not fulfill its original objectives (*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 2018). When used alone, *CoCo* performs at the same level of accuracy as *LLOC* and *HCPL*, and slightly worse than *McCC*. Even when used together with other measures, *CoCo* does not seem to make models more accurate.
**RQ4** How much can source code measures be trusted as understandability predictors? In other words, are predictions based on code measures accurate enough? Do we need to consider additional factors, not involving code, that affect understandability?
The models we found show that there is a correlation between code structural characteristics and code understandability. Nonetheless, the models we found feature a relative absolute error around 30%, on average. The practical usefulness of understandability models featuring 30% error can ultimately be evaluated only subjectively by practitioners, on the basis of their specific needs. However, the prediction error of our models is large enough to suggest that some research is still needed before we can build really reliable models of understandability. We should identify and measure code-unrelated factors

that affect understandability, but also devise new ways of measuring the characteristics of code that most likely affect code readability and understandability.

### 6.2 Verification

During the experiment sessions, participants could have experienced fatigue or tiredness, which could have affected the time taken to perform tasks. Concerning this issue, none of the participants reported to have experienced fatigue or tiredness.

Another possible confounding factor is the well-known learning effect, which could have made the work in the second session easier than in the first one. Indeed, participants stated that the work in the second session seemed easier. Based on this indication, we analyzed the performance of participants in the two sessions, and we found that there is no evidence that tasks were performed faster in the second session.

We used the time taken to perform code correction tasks as a proxy for code understandability. Consequently, we designed all code correction tasks in such a way that their actual difficulty lay in identifying the problem (which, in turn, required understanding the code), while correcting the code required little time. Participants confirmed that performing corrections and checking them via the available test cases was actually quite easy and fast, once the problem had been understood.

Finally, we interviewed industrial developers from two companies to have their opinions on the study and its validity. The interviewed developers appreciated the idea of correlating code metrics with understandability measures and supported the conclusions of the study. However, they also pointed out that the experimentation should be extended to make it more generally representative. That is, they consider the preliminary results we obtained promising and provided some suggestions, which we report in the future work section of the conclusions.

## 7 Threats to Validity

We here summarize the main threats to the validity of our empirical study.

*Internal validity*. We used several techniques to build the various models, with one or with more independent variables. This is common current practice in model building. In the paper, we report on the various decisions we made in the process (e.g., about the hyperparameters), so the readers can evaluate their appropriateness by themselves.

*External validity*. The number of subjects that participated in the empirical study is too small to provide a sufficient degree of external validity to our empirical study. The students formed a homogeneous sample, so they were representative of a portion of the possible population. Due to the characteristics of the current business environment, the subjects could be considered as representative of junior programmers.

*Construct validity*. Software code understandability was measured via the time needed to correctly complete a maintenance task. Thus, we use a time-related measure, as done in a large part of the literature. Other ways of measuring understandability have been used, e.g., based on the number of correct answers on a questionnaire about software code (see Section 8). Thus, we capture one, time-related aspect of code understandability.

As for result evaluation, we used MAR, MdAR, MR, and MdR, which alleviate some of the most relevant construct validity problems of other accuracy metrics such as MMRE.

## 8 Related Work

A few software readability models have been proposed, with different degrees of variety of types of information, features taken into account, and understandability measures.

Buse and Weimer (2010) defined a readability model based on a number of code features, e.g., the number of identifiers, the number of parentheses, the number of branches, the number of loops. They measure readability by having experiment subjects assess several code snippets via an ordinal scale with values from 1 to 5, where value 1 is associated with the least readable code snippets and value 5 with the most readable ones. Then, the study computes the average score for each snippet and builds the distribution of these averages, which presents a natural cut-off score value at 3.14 between more and less readable snippets. Thus, readability is ultimately evaluated on a binary scale: snippets whose average score is below 3.14 are classified as "less readable" and the others as "more readable." So, Buse and Weimer's quantification of readability is quite different from our quantification of understandability, which is based on time. Their readability model was fairly effective in predicting the perceived readability of short code snippets.

The model by Buse and Weimer was later simplified by Posnett et al. (2011), who used only 3 source code measures, namely LOC, entropy, and HVOL. The accuracy of the new model was evaluated via the Area Under the Curve of ROC curves, on the same dataset as Buse and Weimer. The model by Posnett et al. turned out to be more accurate. Posnett et al. also reevaluated the process by which they defined their model in Posnett et al. (2021). Among the main lessons learned, they reiterated a common finding in predictive models based on source code, i.e., the influence of source code size can by no means be ignored.

Dorn (2012) used an additional set of code snippets and built on the previous models by using 4 categories of features, namely visual, spatial, alignment, and linguistic ones, based on the idea that practitioners read source code on screens, so structural code features alone are not sufficient when assessing readability.

A study by Scalabrino et al. (2018) took into account textual features based on source code lexicon analysis in addition to structural features. Their study takes into account the datasets collected by Buse and Weimer and by Dorn along with a new dataset built based on a new set of code snippets. Empirical findings show that textual features complement structural ones. In particular, models based on structural and textual features are more accurate than the previous ones.

A related paper by Fakhoury et al. (2019) describes an empirical study assessing the quality of three readability models when it comes to evaluating readability improvements. These models are based on source code measures and, in their empirical study, Fakhoury et al. used the values of measures extracted by SourceMeter. Readability changes are identified by analyzing the documentation accompanying software code changes. The paper shows the inability of current readability models in capturing readability improvements and recommends that other characteristics be taken into account to build more accurate models for readability changes.

A recent paper by Scalabrino et al. (2021) describes an extensive study in which 444 evaluations concerning 50 methods were provided by 63 Java professional developers and students. Perceived understandability was measured by asking the empirical study participants whether they understood a code snippet. If so, they were asked to answer three confirmation questions, with the purpose of measuring actual understandability. On the other hand, the independent variables included 121 measures related to code, documentation, and developers. The statistical analysis of the collected data found that none of the code measures was significantly correlated with any understandability proxy based on perceived or actual understandability

evaluations. They also built models based on multiple metrics, using several techniques, including machine learning ones. The obtained models show some discriminatory power in the prediction of code understandability proxies, but with very high Mean Absolute Error.

Compared to the work by Scalabrino et al. (2021), our work shows a better ability of code measures (even single ones) in predicting code understandability, even though the prediction errors of our models are definitely not negligible (see Table 3). This is probably due to the fact that Scalabrino et al. employed developers having quite different skills and experience. While the participants in our experiment were characterized by similar skills and experiences, and provided similar performances, the participants in Scalabrino et alii's experiment yielded very different performances. For instance, for 33 out of the 50 methods involved in the experience, there was at least one participant that provided no correct answers to method comprehension questions, and at least one participant that answered correctly all method comprehension questions.

The data collected by Scalabrino et al. were then re-analyzed by Trockman et al. (2018). Scalabrino et al. evaluated the actual understanding of a method based on how many correct answers were provided to the three questions associated with the method. Trockman et al. introduced a binary response variable "*understood*" for every method, that they consider true if two or more questions were answered correctly, false otherwise. Then, they built binary classifiers based on multiple code variables. They found that some metrics correlate with understandability. The models achieved an average Area Under the ROC Curve of 0.64, which implies some discriminating power, but is too low to provide practically reliable predictions.

The results obtained by Trockman et al. are hardly comparably with ours, given not only the diversity of participants' characteristics and the different natures of the considered understandability indicators, but also the difference of metrics used as independent variables. With respect to the latter point, it is worth noting that none of the metrics that are statistically significant in Trockman et alii's models belongs to the set of metrics we investigated.

The appropriateness of using some of the measures listed in Section 3 for maintainability evaluation was discussed by Ostberg and Wagner (2014): Halstead's measures, *McCC*, *LLOC*, and *MI* (Coleman et al. 1994) were not considered adequate, while *NLE* was considered adequate. Even though we agree with Ostberg and Wagner's considerations on a number of counts, we included these measures in our experiment to practically investigate their degree of (in)adequacy.

As a sort of validation of *CoCo*, 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 2018).

An evaluation of how well *CoCo*can be used to assess code understandability based on actual understandability data was performed by Muñoz Barón et al. (2020). 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*. They found moderate associations in some cases.

The issue of the impact of nesting on source code complexity was addressed in software measurement literature prior to *CoCo*, especially with the goal of overcoming some limitations of *McCC*. 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, such as the ones defined by Piwowarski (1982); Dunsmore and Gannon (1979); Harrison and Magel (1981a, b). 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.

## 9 Conclusions and Future Work

We have investigated the possibility of using software source code measures to build models to predict the understandability of software methods. To this end, we carried out an empirical study, in which we collected data about code understandability, and a set of structural measures, to be used as independent variables.

As a measure of code understandability, we used the time needed to correctly complete some maintenance task on code. Since maintenance tasks involve both understanding and modifying the code, we included in the experimental activity only tasks that required very little time to modify the code, once it had been understood. Even so, the code correction time is actually a measure of code understanding, rather than understandability. In fact, code understanding depends on the understandability of code as well as on the ability of involved developers. To minimize the impact of developers' capability and experience on the maintenance time, we selected a set of developers having similar experience and similar capability. Therefore, our results depend almost exclusively on the properties of code.

The models we obtained in our empirical study indicate that code understandability is correlated with the considered set of structural characteristics.

Given the small size of our experiment, especially as far as the number of participants is concerned, our results should be regarded as preliminary indications, which contribute to shed some light on code understandability, rather than definitive conclusions.

The models found seem to indicate that code understandability depends on structural code properties. Nonetheless, the prediction error is around 30%; also the usage of the recently introduced Cognitive Complexity measure does not seem to capture understandability-related aspects better than "traditional" code measures. This outcome of our study indicates that further research is necessary. To improve code understandability models, we should identify and measure properties of the code understanding process, as well as the characteristics of code that most likely affect code readability and understandability and have not yet been suitably described.

Future work will involve

- experimenting in more complex conditions, e.g., involving a larger set of subjects, a larger set of methods, and different processes (e.g., having more than one developer working on a task, like in Pair Programming).
- the use of more source code measures, as well as non-code measures;
- the investigation of other measures than time-related ones to quantify understandability;
- the search for new code measures that are more representative of code understandability than those used in this paper and the literature.

**Acknowledgements** The authors would like to thank the students that participated in the empirical study, the professionals that participated in the interviews, and Anatoliy Roshka and Gabriele Rotoloni, who developed the tool we used to measure Cognitive Complexity.

**Funding** Open access funding provided by Università degli Studi dell'Insubria within the CRUI-CARE Agreement.

## Declarations

**Conflicts of interest** The authors declare that they have no conflict of interest.

**Replication Package** A replication package is publicly available at http://www.dista.uninsubria.it/supplemental_material/understandability/replication_package.zip.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

## References

GitHub (2022) - json-iterator/java: jsoniter (json-iterator) is fast and flexible JSON parser available in Java and Go. https://github.com/json-iterator. Accessed 29 Sept 2023

GitHub - stleary/JSON-java (2022) A reference implementation of a JSON package in Java. https://github.com/stleary/json-java. Accessed 29 Sept 2023

SourceMeter (2022). https://www.sourcemeter.com/. Accessed 29 Sept 2023

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

Arcuri A, Briand L (2014) A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw Test Verif Reliab 24(3):219–250

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

Buse RPL, Weimer W (2010) Learning a metric for code readability. IEEE Trans Software Eng 36(4):546–558. https://doi.org/10.1109/TSE.2009.70

Campbell GA (2018) Cognitive complexity - a new way of measuring understandability. https://www.sonarsource.com/docs/CognitiveComplexity.pdf. Accessed 29 Sept 2023

Campbell GA (2018) Cognitive complexity: An overview and evaluation. In: Proceedings of the 2018 international conference on technical Debt, pp 57–58

Carver JC, Jaccheri L, Morasca S, Shull F (2010) A checklist for integrating student empirical studies with research and teaching goals. Empir Softw Eng 15(1):35–59. https://doi.org/10.1007/s10664-009-9109-9

Chen ET (1978) Program complexity and programmer productivity. IEEE Trans Software Eng 4(3):187–194. https://doi.org/10.1109/TSE.1978.231497

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

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

Dorn J (2012) A general software readability model. Department of Computer Science, Master's thesis, University of Virginia, Charlottesville, Virginia

Dunsmore HE, Gannon JD (1979) Data referencing: An empirical investigation. Computer 12(12):50–59. https://doi.org/10.1109/MC.1979.1658576

Fakhoury S, Roy D, Hassan SA, Arnaoudova V (2019) Improving source code readability: theory and practice. In: Guéhéneuc, Y, Khomh, F, Sarro F (eds.) Proceedings of the 27th international conference on program comprehension, ICPC 2019, Montreal, QC, Canada, May 25-31, 2019, pp 2–12. IEEE / ACM. https://doi.org/10.1109/ICPC.2019.00014

Fenton NE, Bieman JM (2014) Software metrics: a rigorous and practical approach, third edition. Chapman & Hall/CRC innovations in software engineering and software development series. Taylor & Francis. https://books.google.es/books?id=lx_OBQAAQBAJ. Accessed 29 Sept 2023

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) pp 175–186. IEEE

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) pp 311–322. IEEE

Glass RL (2001) Frequently forgotten fundamental facts about software engineering. IEEE Softw 18(3):112

Halstead, MH (1977) Elements of software science. Elsevier North-Holland

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

Harrison WA, Magel KI (1981b) A complexity measure based on nesting level. SIGPLAN Not 16(3):63–74. https://doi.org/10.1145/947825.947829

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) pp 30–39. IEEE

Hofmeister JC, Siegmund J, Holt DV (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, pp 217–227. IEEE Computer Society. https://doi.org/10.1109/SANER.2017.7884623

Howatt JW, Baker AL (1989) Rigorous definition and analysis of program complexity measures: An example using nesting. J Syst Softw 10(2):139–150. https://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) pp 1–6. IEEE

Li EY (1987) A measure of program nesting complexity. In: 1987 AFIPS national computer conference (NCC) pp 531–538. AFIPS

McCabe TJ (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, pp 25–35. IEEE

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, pp 44–55. IEEE Computer Society, Washington, DC, USA. https://doi.org/10.1109/ESEM.2009.5316048

Muñoz Barón M, Wyrich M, Wagner S (2020) 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

Oliveira D, Bruno R, Madeiral F, Castor F (2020) Evaluating code readability and legibility: An examination of human-centric studies. In: IEEE International conference on software maintenance and evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020, pp 348–359. IEEE. https://doi.org/10.1109/ICSME46990.2020.00041

Oman P, Hagemeister J (1992) Metrics for assessing a software system’s maintainability. In: Proceedings conference on software maintenance 1992, pp 337–338. IEEE Computer Society

Ostberg JP, 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, pp 32–37. IEEE

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 Software Eng 46(4):442–462. https://doi.org/10.1109/TSE.2018.2863303

Piwowarski P (1982) A nesting level complexity measure. ACM SIGPLAN Notices 17(9):44–50. https://doi.org/10.1145/947955.947960

Posnett D, Hindle A, Devanbu PT (2011) A simpler model of software readability. In: van Deursen A, Xie T, Zimmermann T (eds.) Proceedings of the 8th international working conference on mining software repositories, MSR 2011 (Co-located with ICSE) Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings, pp 73–82. ACM. https://doi.org/10.1145/1985441.1985454

Posnett D, Hindle A, Devanbu PT (2021) Reflections on: A simpler model of software readability. ACM SIGSOFT Softw Eng Notes 46(3):30–32. https://doi.org/10.1145/3468744.3468754

R core team (2015) R: a language and environment for statistical computing

Salvaneschi G, Amann S, Proksch S, Mezini M (2014) An empirical study on program comprehension with reactive programming. In: Cheung S, Orso A, Storey MD (eds.) Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, (FSE-22) Hong Kong, China, November 16 - 22, 2014, pp 564–575. ACM. https://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 Software Eng 47(3):595–613. https://doi.org/10.1109/TSE.2019.2901468

Scalabrino S, Linares-Vásquez M, Oliveto R, Poshyvanyk D (2018) A comprehensive model for code readability. J Softw Evol Process 30(6). https://doi.org/10.1002/smr.1958

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

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 MP, 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, p 24. ACM. https://doi.org/10.1145/2393596.2393624

Trockman A, Cates K, Mozina M, Nguyen T, Kästner, C, Vasilescu, B (2018) Automatically assessing code understandability reanalyzed: combined metrics matter. In: 2018 IEEE/ACM 15th international conference on mining software repositories (MSR) pp 314–318. IEEE

Vargha A, Delaney HD (2000) A critique and improvement of the cl common language effect size statistics of mcgraw and wong. J Educ Behav Stat 25(2):101–132

Welker KD, Oman PW, Atkinson GG (1997) Development and application of an automated source code maintainability index. J Softw Maint Res Pract 9(3):127–159

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

**Publisher's Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

## Authors and Affiliations

**Luigi Lavazza[1] · Sandro Morasca[1] · Marco Gatto[1]**

Sandro Morasca
sandro.morasca@uninsubria.it

Marco Gatto
marcogatto.1997@hotmail.it

[1] Dipartimento di Scienze Teoriche e Applicate, Università degli Studi dell'Insubria, Varese, Italy