From Code Complexity Metrics to Program Comprehension

Dror G. Feitelson
feit@cs.huji.ac.il
The Hebrew University
Jerusalem, Israel

ABSTRACT

Developers need to read and understand a lot of existing code during the course of their work. But understanding code written by others is hard. It is common to blame this on the code's "complexity", possibly as measured by metrics like McCabe's Cyclomatic Complexity (MCC). However, commonly used metrics are often simplistic and based on intuition rather than on empirical evidence. This can be much improved by using experiments to identify and quantify the effect of coding constructs on code comprehension. However, the way individual human developers perceive the code may actually be more important than the characteristics of the code itself: for example, one developer may misunderstand variable names chosen by another. The diversity of humans implies that a comprehensive code complexity metric may therefore be impossible to achieve. But experimental studies of code comprehension can nevertheless shed light on the limitations of code complexity metrics, and lead to better understanding of the cognitive processes involved in program comprehension.

CCS CONCEPTS

• **General and reference** → **Experimentation**; **Metrics**; • **Software and its engineering** → **Software organization and properties**.

KEYWORDS

Experiment, Code complexity, Code comprehension

ACM Reference Format:
Dror G. Feitelson. 2023. From Code Complexity Metrics to Program Comprehension. In *Comm. ACM 66(5), pp. 48–57, May 2023.* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

Code is hardly ever developed from scratch. Rather, new code typically needs to integrate with existing code, and depends on using existing libraries. Two recent studies have found that developers spend 58% and 70% of their time on average comprehending code, and only some 5% actually editing [31, 51]. This implies that reading and understanding code is very important, both as an enabler of development, and as a major cost factor during development.

Comm. ACM 66(5), pp. 48–57, May 2023,

But as anyone who tries to read code can attest, it is hard to understand code written by others. This is commonly attributed, at least in part, to the code's "complexity": the more complex the code, the harder it is to understand, and by implication, to work with. Identifying and dealing with complexity is considered important because the code's complexity may slow developers down, and may even cause them to misunderstand it, possibly leading to programming errors. Conversely, simplicity is often extolled as vital for code quality.

To gain a sound understanding of code complexity and its consequences, we need to operationalize this concept. This means we need to come up with ways to characterize it, ideally in a quantitative manner. And indeed many metrics have been suggested for code complexity. Such metrics can then be used for either of two purposes. In industry, metrics are used to make predictions regarding code quality and development effort. This can then feed into decision-support systems that help managers steer the project [15]. In academia, metrics can be used to characterize the code and better understand how developers interact with the code on which they work. This can then contribute to modeling the cognitive processes involved in software development.

While the concept of code complexity has intuitive appeal, and literally hundreds of metrics have been proposed, it has been difficult to find agreed metrics that quantify code complexity effectively. One reason may be that proposed metrics typically focus on a single narrow attribute of complexity, and they are often not validated by empirical evidence showing an effect on code comprehension. As a result it is not very surprising that they do not enable good predictions. An alternative approach can be to use an incremental process, guided by empirical evaluations, to try and create more comprehensive metrics. However, as we show below, this approach too faces many difficulties: partly due to the complexity of the concept of code complexity itself, and partly due to its interactions with the humans involved in code comprehension. The conclusion is that code complexity is only one of many factors affecting code comprehension, and perhaps not the major one.

2 METRICS FOR CODE COMPLEXITY

Various code attributes have been suggested as contributing to code complexity. For example, the use of wild goto statements creates spaghetti code which is hard to follow [10, 36]. The use of pointers and recursion are also thought to cause difficulties [47]. Various code smells also reflect excessive complexity, ranging from long parameter lists to placing all the code in a single god-class with no modular design [43]. Finally, just having a large volume of code also adds to the difficulty of understanding it all [18]. Note, however, that understanding code is different from understanding a system as a whole, where the main concern is understanding the architecture

[27]. Our focus is on understanding the code as a text conveying instructions.

Given code attributes that are hypothesized to contribute to complexity, one can define code complexity metrics to measure them. Such metrics use static analysis to identify and count the problematic code features. Over the years extensive research has been conducted on code complexity metrics. Much of this research concerned the direct utility of such metrics, for example as predictors of defects [9, 16, 30]. While such correlations have indeed been found (e.g. [40]), it also appears that no single metric is universally applicable, and in fact all metrics have a poor record of success [11, 13, 33, 34, 39]. Another result is that process metrics are actually better at predicting defects than code metrics, because problematic code tends to stay so [37].

Indeed, a perplexing observation concerns the apparent lack of progress in defining software metrics. In their 1993 book on software metrics, Shepperd and Ince discuss the three most influential software metrics of the time [44]. These were Halstead's "software science" metrics [19], McCabe's cyclomatic complexity [28], and Henry and Kafura's information flow metric [21], which were 12–19 years old at the time. Were they to write the book today, a full 30 years later, they would probably have chosen the same three metrics as the most influential (perhaps adding Chidamber and Kemerer's metric suite [8], which is specifically targeted at object-oriented designs). This is surprising and disillusioning given the considerable criticism leveled at these metrics over the years, including in Shepperd and Ince's book.

Let us use McCabe's cyclomatic complexity (MCC) as an example. This is perhaps the most widely cited complexity metric in the literature[1]. It is the default go-to whenever "complexity" is discussed. For example, one of the metrics in Chidamber and Kemerer's metric suite is "weighted methods per class" [8]. In its definition the weighting function was left unspecified, but in practice it is usually implemented as the methods' MCC.

The essence of MCC is simply the number of branching instructions in the code plus 1, counted at the function level. This reflects the number of independent paths in the code, and McCabe suggested that functions with MCC above 10 may need to be simplified, lest they be hard to test [28]. Over the years, many have questioned the definition of MCC. For starters, does it really provide any new information? Several studies have shown a very strong correlation between MCC and LOC (lines of code), implying that MCC is more of a size metric than a complexity metric [18, 23, 45]. But others have claimed that MCC is indeed useful, and explained the correlation with LOC as reflecting an average when large amounts of code are aggregated. If a finer resolution of individual methods is observed, there is a wide variation of MCC for functions of similar length [26].

Other common objections concern the actual counting of branching instructions. One issue is exactly which constructs to count. McCabe originally counted the basic elements of structured programming in Fortran, for example if-then-else, while, and until. He also noted that in compound predicates the individual conditions should be counted, and that a case statement with N branches should be counted as having $N - 1$ predicates. Soon after it was suggested that nesting should also be taken into consideration [20, 35]. Vinju and Godfrey suggest to add elements like exit points (break, continue) and exception handling (try, throw) to the calculation of complexity [49]. Campbell went even farther, and created a much more comprehensive catalog of constructs, including nesting, try-catch blocks, and recursion, which should all be counted [7].

Given the zoo of constructs that may affect code complexity, another issue is the weights one gives to different constructs. Intuitively, it doesn't seem right to give the same level of importance to a while loop and a case statement. There has been surprisingly little discussion of this issue. Campbell, for example, does not count individual cases at all, and gives added weight to each level of nesting [7]. Shao and Wang suggest that if a sequence has weight 1, an if should have weight 2, a loop weight 3, and parallel execution weight 4 [42]. But this assignment of weights is based only on intuition, which is not a scientifically valid way to devise a code complexity metric [29].

The common methodology to assess the utility of complexity metrics is based on correlations with factors of interest [41]. In an extensive recent study, Scalabrino et al. show that no individual metric of the 121 they checked captures code understandability [39]. But more positive results are sometimes obtained when metrics are combined [39, 48]. For example, Muñoz Barón et al. have shown that Campbell's metric has a positive correlation with the time needed to understand code snippets [32]. Buse and Weimer created a "readability" model (which actually reflects perceived understandability, as this is what was asked of evaluators) based on 19 code features, including line and identifier length, indentation, and numbers of keywords, parentheses, assignments, and operators [6].

It therefore seems that at least part of the problem with code complexity metrics may be their fragmentation and rigidity. Maybe we can achieve better results by combining individual metrics in a systematic manner. In software development the iterative and incremental approach – with feedback from actual users – is widely accepted as the way to make progress when we cannot define everything correctly in advance. Why not use the same approach to derive meaningful metrics for software?

3 EXPERIMENTAL EVALUATIONS

If we want to understand the difficulties in comprehending code, and to parameterize code complexity metrics in a meaningful way, we need to study how developers think about code [22]. A common approach for research on people's cognitive processes is controlled experiments. In the natural sciences experiments are akin to posing a question to nature: we set up the conditions, and see what nature does. In empirical software engineering, and specifically when studying code comprehension, we perform the experiments on developers. The developers are assigned tasks which require some code to be understood. Example tasks include figuring out the output of a code snippet on some specific input, finding and fixing a bug in the code, and so on. By measuring the time to complete the task, and the correctness of the solution, we get some indication of how hard it was [14]. And if the difference between the experimental conditions was limited to some specific attribute of the code, we

[1] McCabe's 1976 paper which introduced MCC [28] had 8806 citations on Google Scholar as of 22 March 2023 (304 in 2022, probably not the final count, 352 in 2021, 393 in 2020, 457 in 2019, 433 in 2018, and so on).

Table 1: *Characteristics of study 1 [1].*

Number of subjects	220
Subjects status	professional developers
Experimental task	output printed by code
Total code snippets	40
Code snippets source	written for experiment
Snippets for each subject	11–14
Snippets selection	random by group
Snippets order	random
Setting	custom-built Internet site
Performance comparison	within subject

can gain information on the effect of this attribute — namely on its contribution to the code's complexity.

Experimental evaluations like this are not very common [46], possibly in part due to their human element. When we design such experiments we must think and reflect about who the participants will be, and whether the results may depend on their experience. For example, "born Python" developers are accustomed to implicit loops in constructs like list comprehension, which are problematic for developers used to more explicit languages. Another common issue to consider is whether students may be used [3, 12]. Comprehensive results require the use of all relevant classes of developers, even if not necessarily in the same study.

The following sections present three recent case studies of such research, and what we can learn from them about the relationship between code complexity and code comprehension. This is by no means a comprehensive survey. These studies are just isolated examples of what can be done, mainly illustrating how much we still need to learn. At the same time, they also indicate that maybe the quest for a comprehensive complexity metric is hopeless, both because there are so many conflicting factors that affect complexity, and because the code itself may not be the most important factor affecting comprehension.

4 STUDY 1: CONTROLLED EXPERIMENTS ON LOOPS

The first example is a study motivated by the questions about MCC alluded to above, which asks whether loops and if statements should be given the same weight. As we noted Shao and Wang have already suggested that different weights be given to different constructs [42]. But while intuitively appealing, this proposal was not backed by any evidence that these are indeed the correct weights.

To place such proposals on an empirical footing we designed an experiment where subjects needed to understand short code snippets [1]. These code snippets were written specifically for the experiment, and different snippets used different control structures. Technical attributes of the experiment are listed in Table 1.

To ensure that comparisons are meaningful all the code snippets had exactly the same functionality: to determine whether or not an input number x was in any of a set of number ranges. This can be expressed using nested if statements that compare x with the endpoints of all the ranges. Alternatively it is possible to create one large compound conditional which contains a disjunction of conjunctions for the endpoints of each range. One can also use a

```
string result = "no...";
if (x >= 10) {
    if (x <= 20) {
        result = "yes!";
    }
    else {
        if (x >= 30) {
            if (x <= 40) {
                result = "yes!";
            }
        }
    }
}
```

```
string result = "no...";
if (((x >= 10) && (x <= 20)) ||
    ((x >= 30) && (x <= 40))) {
    result = "yes!";
}
```

```
string result = "no...";
int ends[][] = {{10, 20}, {30, 40}};
for (int i=0; i<2; i++) {
    if ((x >= ends[i][0]) &&
        (x <= ends[i][1])) {
        result = "yes!";
    }
}
```

```
string result = "no...";
for (int i=0; i<2; i++) {
    if ((x >= (2*i+1)*10) &&
        (x <= (2*i+2)*10))) {
        result = "yes!";
    }
}
```

Figure 1: *Example code snippets which use different constructs to express the same functionality. From the top: nested ifs, compound conditional, loop on array, loop with arithmetic on loop index.*

loop on the ranges, and compare with the endpoints of one range in each iteration. The endpoints can be stored in an array, or, if they are multiples of a common value, they can be derived by arithmetic expressions on the loop index. Examples of code snippets implementing these four approaches are shown in Figure 1.

The results obtained in the experiment for these four options are shown in Table 2 (in the full experiment additional structures were also studied, e.g. using negations, which are not shown here). One can see that finding the outcome when the code contained loops took significantly more time, and led to significantly more errors — roughly by a factor of 2. This indicates that the methodology works: we can design experiments which uncover differences in the

Table 2: *Results of comparison of loops with ifs.*

Code snippet	*Time [s]*	*Err. rate*
Nested ifs	16.7±9.9	0.175
Compound if	23.5±10.4	0.082
Loop on array	35.0±14.7	0.450
Loop with arith.	46.6±19.0	0.325

Table 3: *Variations between code snippets that compare for loops*

Version	*Init*	*Comp*	*End*	*Step*
lp0	0	<	n	++
lp1	0	<=	n-1	++
lp2	0	<	n-1	++
lp3	1	<	n	++
lp4	1	<	n-1	++
lp5	n-1	>=	0	--
lp6	n-1	>	0	--

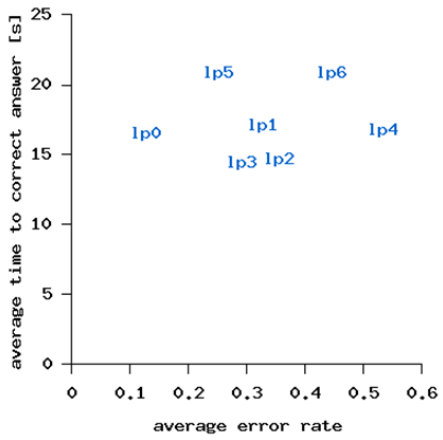


Figure 2: *Results of experiment comparing the understanding of variations on the canonical for loop.*

performance of developers dealing with different code constructs. And if we want to create a better version of MCC, giving loops double the weight of ifs is a reasonable initial estimate, more so than giving them equal weights. However this cannot be considered the final say on the matter. Many additional experiments with different loop structures and ifs are needed.

In addition to comparing different constructs, the experiment also included 7 code snippets that compared variations on for loops. The canonical for loop is for (i=0; i<n; i++). But the intialization, the end condition, and the step may be varied. In the experiment we looked at 6 such variations, as described in Table 3. The task for the experimental subjects was to list the loop index as it would be printed in each iteration. The results are shown in Figure 2. This is a scatter plot showing how experiment participants performed on the seven versions of the loop. The horizontal dimension represents the error rate, namely the fraction of wrong answers we received. The vertical dimension represents the average time taken to provide correct answers. The time of incorrect answers is not used.

The canonical for loop, represented by lp0, is the leftmost point. Its coordinates indicate that it takes about 17 seconds to understand, and around 10% of respondents get it wrong. The next 3 versions each have one simple variations. Loop lp1 is actually equivalent to the canonical loop, but expressed differently: the end condition is <=n-1 instead of <n. Loops lp2 and lp3 have a difference of one at either end. All these loop take about the same time as the canonical loop, but suffer from about 20% more errors. Loop lp4 has two variations from the canonical loop: it both starts at 1 and ends at <n-1. It again takes approximately the same time, but now more than half of the respondents make mistakes. The conclusion from all this is that many experimental subjects apparently didn't notice the changes: seeing a for they expected to see a canonical for loop, and answered as if it was, thereby making a mistake. In other words, the metric of error rate is not equivalent to the metric of time to correct answer. Error rate may indeed reflect difficulty, but this is confounded with a "surprise factor", where the error reflects a clash between expectations and reality. So the important implication is that *difficulties in understanding are not just a property of the code.* They may also depend on the interaction between the code and the person reading it.

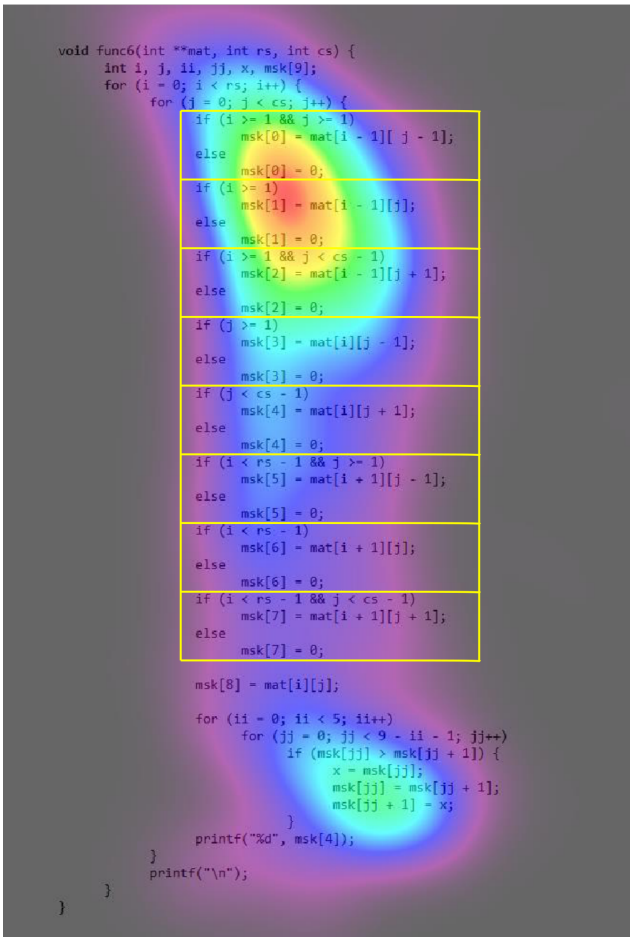
The last two loops, lp5 and lp6, are different in that they count down instead of counting up. This increases the time to correct answer by about 4 seconds, or 25%, which can be interpreted as reflecting extra cognitive effort to figure out exactly what is going on. It also increases the error rate by around 10 percentage points relative to the equivalent up-counting loops. These results provide yet another example of the failings of metrics like MCC: in terms of MCC, loops counting up and loops counting down are equivalent. But for human beings they are not.

5 STUDY 2: USING EYE TRACKING TO STUDY CODE REGULARITY

An additional problem with many common code complexity metrics is that they ignore repetitions in the code. More than 30 years ago Weyuker noted that conjugating two copies of a program is expected to be easier to understand than twice the effort of understanding one copy [50]. More recently, Vinju and Godfrey made a similar empirical observation: they saw that repeated code is easier than implied by its MCC [49]. But how much easier? And how can we account for such effects in code complexity metrics?

Our second example is a study which addresses these issues using eye tracking. The participants in the study are again given code that they need to understand, but as they read it, we observed how they divide their attention across repetitions in the code [25]. Technical attributes of the experiment are listed in Table 4.

The study was based on two functions, each with two versions: a regular version with repeated structures and a non-regular version without such repeated structures. For example, one of the functions performed an image processing task of replacing each pixel with the median of its 3×3 neighborhood. The problem is that boundary pixels do not have all 8 neighbors. The regular version solves this by checking for the existence of each neighbor, one after the other, thereby leading to a repeated structure. The non-regular version, in contradistinction, first copies the image into a larger matrix



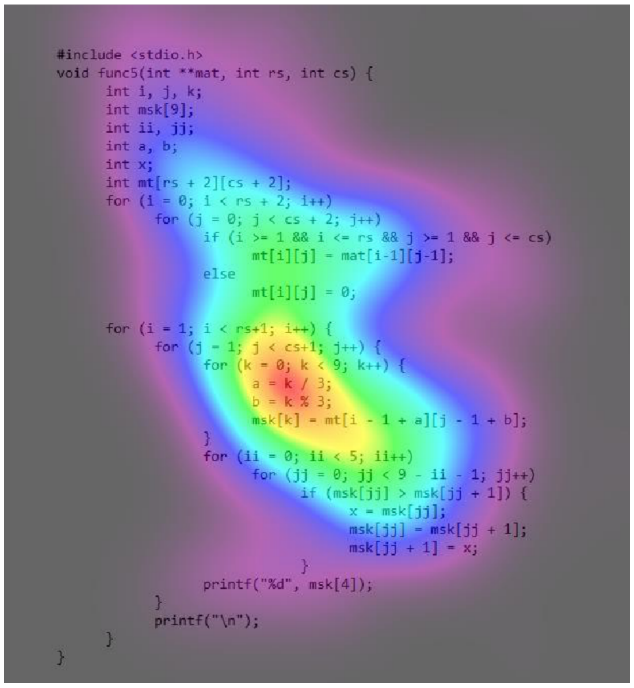


Figure 3: *Heatmap of visual attention on regular code (left, with repeated structures marked) and non-regular code (above). Reprinted with permission from Springer [25] ©2017*

Table 4: *Characteristics of study 2 [25].*

Number of subjects	20
Subjects status	students
Experimental task	understand code
Total functions used	2
Functions source	written for experiment
Versions of each function	2
Assigning functions/versions	random one each
Setting	1:1 sessions with eye tracker
Performance comparison	between subjects

leaving a boundary of 0 pixels around it. It can then traverse the image without worrying about missing neighbors. Importantly, both approaches are reasonable and used in practice. Thus the code was realistic despite being written for the experiment.

Because of the repetitions, regular code tends to be longer and have a higher MCC. For example, in the image processing task cited above, the regular version was 49 lines long and had an MCC of 18, while the non-regular one had only 33 lines of code and an MCC of 13. But the experiments showed that the regular version was easier to understand [24]. This was verified both by metrics like time to correct answer and by subjective ratings. Using a 5-point scale from "very easy" to "very hard", subjects rated the regular version as "easy", "moderate", or "hard", at the same time rating the non-regular one as "moderate", "hard", or "very hard". The implication is that MCC indeed fails to correctly reflect the complexity of this code.

Our hypothesis in the experiment was that the regular version is easier because once you understand the initial repeated structure, you can leverage this understanding for the additional instances. This led to the prediction that less and less attention would be devoted to successive repetitions. But when we read, the eyes do not move continuously. Rather, they fixate on certain points for sub-second intervals, and jump from one fixation point to the next. Cognitive processes like understanding occur during these fixations. So we can check our prediction by using an eye tracker to measure the number of fixations in different parts of the code and their total duration. A heatmap of where subjects were found to spend their time is shown in Figure 3. Obviously in the regular version they indeed spend much more time on the initial repetitions, the hardly any on the later ones. Then they return to pay attention to the final loops that calculates the median. In the non-regular versions most attention is focused on the nested loops that scan the image and collect data from the neighboring pixels.

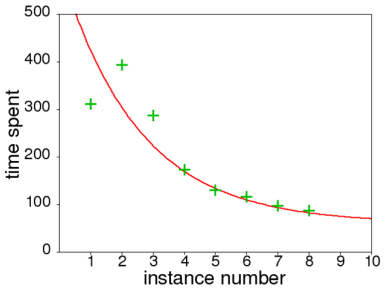


Figure 4: *Measurements and model of attention given to structural repetitions in regular code.*

The figure also indicates the "areas of interest" we defined comprising the 8 repetitions of the regular structure. Once defined, we can sum the total time spent in each such area. The results are shown in Figure 4. Fitting a model to these measurements indicates that an exponentially decreasing function provides a good fit. Specifically, the model indicates that the time invested is reduced by 40% with each additional instance. This suggests that when computing complexity metrics the weight given to each successive instance should be reduced by 40% relative to the previous one. For example, the complexity of k repetitions of a block of code with complexity C would not be kC but

$$comp = \sum_{i=1}^{k} 0.6^{i-1} C$$

Again, we cannot claim that this is the final word on this issue. However, it does show how code complexity metrics can be improved and better aligned with developer behavior.

The more important conclusion, however, is that code complexity is not an absolute concept. The *effective complexity of a block of code may depend on code that appeared earlier*. In other words, complexity depends on context. It is not simply additive, as assumed by MCC and other complexity metrics.

6 STUDY 3: THE EFFECT OF VARIABLE NAMES

A basic problem in understanding code is to identify the domain. For example, are the strings manipulated by this function the names of people? Or maybe addresses? Or the contents of email messages? The code itself seldom makes this explicit. But the names of classes, functions, and variables do. This is what we mean when we demand that developers "use meaningful names". Names are therefore extremely important for code comprehension [4, 17, 38]. It is the names that enable us to form an understanding of what the code is about, and subsequently, what it does.

Our third example is a study that set out to quantify the importance of names. To do so, participants in the study were given code in either of two versions: one with full variable names, or another with meaningless names [2]. In both cases they were required to figure out what the code does. Technical attributes of the experiment are listed in Table 5.

For this study it was important the the names reflect real practice. We therefore scanned around 200 classes from 30 popular (with at least 10,000 stars) Java utility packages on GitHub. Utility packages

Table 5: *Characteristics of study 3 [2].*

Number of subjects	9
Subjects status	professional developers
Experimental task	understand code
Total functions used	6
Functions source	popular GitHub projects
Versions of each function	2 (4 incl. partial names)
Assigning versions	random
Setting	1:1 sessions think aloud
Performance comparison	between subjects

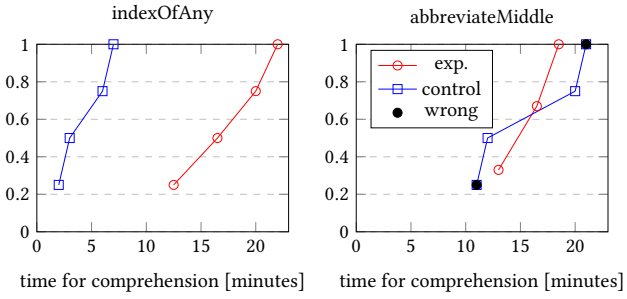


Figure 5: *Examples of results from the variable names study. Reprinted with permission from IEEE [2] ©2017*

were used to avoid domain knowledge issues; it was assumed that any developer has the knowledge to understand functions that perform things like string manipulation. From this vast pool a total of 12 functions were selected, based on considerations of length and perceived difficulty. A pilot study was then conducted to ensure the functions were suitable for use in the experiment. This narrowed the field down to 6 final functions.

To study the importance of the names in the functions we replaced them with consecutive letters of the alphabet: a, b, c, and so on, as many as were needed, in order of appearance. Half of the experimental subjects received this treatment. The other half received the functions with their original variable names as a control. The study was conducted with professional developers using 1:1 sessions in which they explained their thoughts about what they are trying to do. If they did not reach a conclusion within 10 minutes, some of the names (either the function parameters or its local variables) were revealed. We expected that seeing the variable names to begin with, or getting them in the process, will aid comprehension.

Two examples of the results are shown in Figure 5. These are cumulative distribution functions (CDFs) of the time till an answer was given. The horizontal axis represents time in minutes. The vertical axis is the cumulative probability to receive an answer within a certain time. The left-hand graph, showing the results for the function indexOfAny, matches our expectations. As we can see, the subjects in the control group, who received the function with the original variable names, took between 2 and 7 minutes to give an answer. Those in the treatment group, where the variables were renamed to a, b, c, etc., took 12–22 minutes. Much longer, because the additional information provided by the variable names

is missing and needs to be reconstructed from the functionality of the code. Similar results, with somewhat smaller gaps between the times, were observed in two other functions of the 6 used in the experiment.

But the results of the second example function, abbreviateMiddle, are different. In this case the distributions of times to answers are overlapping — it took approximately the same time whether or not the names were provided. Moreover, two of the subjects misunderstood the code and claimed it does something different from what it really does. And both those who made mistakes had received the original code with the names. Similar results were obtained for another two functions. In both, the distributions of times were overlapping, and in one there were another 2 mistakes, again both by subjects who had received the full-names version.

The conclusion from this experiment is that names do not necessarily help comprehension. Moreover, it appears that there are situations where names are misleading, to the degree that they are worse than having meaningless names such as consecutive letters of the alphabet. How can this be? In the few examples we saw, one problem was using general non-specific names. For example a function signature contained two string parameters, one called str, and an integer called length. But length of what? It turns out that it specified the target length of the output, but this was not apparent from the names. Another problem was the use of synonyms. For example, in a function involving two strings, the length of one was called length and the length of the other was called size. It is easy to get confused which is which, because the names do not contain any distinguishing information.

Interestingly, type names can also cause problems. One of the examples was a function which replaced characters from one set with characters from another. But the sets were passed to the function as strings instead of as arrays of characters. From the computer's point of view there is no difference, and it works. But for human readers the "string" signal was so strong they thought the function replaces one string for the other, instead of individual characters.

An important point is that we do not think that misleading names were used on purpose. Recall that the code comes from highly popular open source projects, and it is safe to assume that the developers who wrote it were trying to use meaningful names. But apparently it is not so easy to find meaningful names, partly because *names that are meaningful to one developer can be misleading to another*. This implies that understanding code can be impaired by a mismatch between the developer who wrote it and the developer who is trying to understand it, through no fault of the code itself.

7 UNDERSTANDING CODE COMPREHENSION AND COMPLEXITY

Selecting code metrics is often driven by inertia — we continue to use what has been used before. But such metrics are oftentimes very simplistic and based on intuition, and rigorous evaluations to see if they work usually find that they do not perform well (e.g. [34, 39]). Despite this, literally hundreds of papers each year continue to use MCC as it was defined more than 45 years ago. At the same time there have been only a handful of attempts to find empirical support for more comprehensive metrics.

The quest for better complexity metrics is driven by an agenda to explain the difficulties of interacting with code using the code's properties. Computer scientists are trained to solve big problems by dividing them into more solvable sub-problems. So a natural place to start with the riddle of how code is comprehended is with the most static and well-defined parts, namely individual programming constructs and their composition in functions. This can be followed by models of module comprehension and even full projects.

The problem is that analyzing the code has its limitations. As Fred Brooks wrote, "The programmer, like the poet, works only slightly removed from pure thought-stuff" [5]. And indeed, programming is the art of ideas, abstractions, and logic. But the code itself is written at a lower level, of concrete instructions to be carried out by a computer. The difficulty in comprehending code is thus the difficulty to reconstruct the ideas and abstractions — the art and the "thought stuff" — when all you have available are the instructions. Moreover, this has to be done subject to human differences. The expectations of the developer reading and trying to understand the code may be incompatible with the practices of the developer who wrote it. Such discrepancies are by definition beyond the reach of code complexity metrics, and place a limit on the aspiration to analyze code and deduce its objective comprehensibility.

An alternative framing is therefore not to seek a comprehensive complexity metric that will allow us to predict the difficulty of interacting with a given body of code, but to seek a deeper understanding of *the limitations* of code complexity metrics. The methodology is the same: to work diligently on experiments to isolate and estimate the influence of different constructs; to conduct further experiments that can also elucidate the interactions between them; to acknowledge and study other factors, such as how the code is presented and the effect of the experience and background of developers; and to replicate all this work multiple times to increase our confidence in the validity of the results.

The three studies described above provide examples of such work. Study 1 shows how individual structures can be studied in a quantitative manner. Study 2 concerns how to combine such results, and undermines the idea of just summing the counts of constructs as is commonly done. Study 3 is a small step toward showing that the code may not be the main factor at all. There are many additional factors that need to be studied and evaluated, for example the dependencies on rapidly evolving third-party libraries. Additional experiments will also need to take into account all the different classes of developers: experienced professionals, novices, even end-users. Observational studies "in the wild" can be used to verify the relevance of experimental results in reality.

The results will be messy, in the sense that myriad conflicting effects can be expected. But this is what makes the code-developer interaction more interesting, challenging, and important to understand. And it also suggests interesting opportunities for collaborations between computer scientists and cognitive scientists. Understanding code does not depend only on the code — it also depends on the brain. As computer scientists we do not have all the necessary background and accumulated knowledge about cognitive processes. But psychologists and cognitive scientists have been studying such phenomena for decades. If we really want to understand code comprehension, we need to collaborate with them.

ACKNOWLEDGMENTS

The actual work of designing and running the experiments described above was done by Shulamyt Ajami (loops study), Ahmad Jbara (code regularity study), and Eran Avidan (naming study). Funding was provided by the ISRAEL SCIENCE FOUNDATION (grants 407/13 and 832/18).

REFERENCES

[1] S. Ajami, Y. Woodbridge, and D. G. Feitelson, "*Syntax, predicates, idioms — what really affects code complexity?*" *Empirical Softw. Eng.* **24(1)**, pp. 287–328, Feb 2019, DOI: 10.1007/s10664-018-9628-3.

[2] E. Avidan and D. G. Feitelson, "*Effects of variable names on comprehension: An empirical study*". In 25th *Intl. Conf. Program Comprehension*, pp. 55–65, May 2017, DOI: 10.1109/ICPC.2017.27.

[3] V. R. Basili and M. V. Zelkowitz, "*Empirical studies to build a science of computer science*". *Comm. ACM* **50(11)**, pp. 33–37, Nov 2007, DOI: 10.1145/1297797.1297819.

[4] S. Blinman and A. Cockburn, "*Program comprehension: Investigating the effects of naming style and documentation*". In 6th *Australasian User Interface Conf.*, pp. 73–78, Jan 2005.

[5] F. P. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

[6] R. P. L. Buse and W. R. Weimer, "*Learning a metric for code readability*". *IEEE Trans. Softw. Eng.* **36(4)**, pp. 546–558, Jul/Aug 2010, DOI: 10.1109/TSE.2009.70.

[7] A. Campbell, "*Cognitive complexity: A new way of measuring understandability*". URL https://www.sonarsource.com/docs/CognitiveComplexity.pdf, 2016.

[8] S. R. Chidamber and C. F. Kemerer, "*A metric suite for object oriented design*". *IEEE Trans. Softw. Eng.* **20(6)**, pp. 476–493, Jun 1994, DOI: 10.1109/32.295895.

[9] D. Cotroneo, R. Pietrantuono, and S. Russo, "*Testing techniques selection based on ODC fault types and software metrics*". *J. Syst. & Softw.* **86(6)**, pp. 1613–1637, Jun 2013, DOI: 10.1016/j.jss.2013.02.020.

[10] E. W. Dijkstra, "*Go To statement considered harmful*". *Comm. ACM* **11(3)**, pp. 147–148, Mar 1968, DOI: 10.1145/362929.362947.

[11] S. Fakhoury, D. Roy, S. A. Hassan, and V. Arnaoudova, "*Improving source code readability: Theory and practice*". In 27th *Intl. Conf. Program Comprehension*, pp. 2–12, May 2019, DOI: 10.1109/ICPC.2019.00014.

[12] D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo, "*Empirical software engineering experts on the use of students and professionals in experiments*". *Empirical Softw. Eng.* **23(1)**, pp. 452–489, Feb 2018, DOI: 10.1007/s10664-017-9523-3.

[13] J. Feigenspan, S. Apel, J. Liebig, and C. Kästner, "*Exploring software measures to assess program comprehension*". In *Intl. Symp. Empirical Softw. Eng. & Measurement*, pp. 127–136, Sep 2011, DOI: 10.1109/ESEM.2011.21.

[14] D. G. Feitelson, "*Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension*". *Empirical Softw. Eng.* **27(6)**, art. 123, Nov 2022, DOI: 10.1007/s10664-022-10160-3.

[15] N. E. Fenton and M. Neil, "*Software metrics: Successes, failures and new directions*". *J. Syst. & Softw.* **47(2–3)**, pp. 149–157, Jul 1999, DOI: 10.1016/S0164-1212(99)00035-7.

[16] N. E. Fenton and M. Neil, "*A critique of software defect prediction models*". *IEEE Trans. Softw. Eng.* **25(5)**, pp. 675–689, Sep/Oct 1999, DOI: 10.1109/32.815326.

[17] E. M. Gellenbeck and C. R. Cook, "*An investigation of procedure and variable names as beacons during program comprehension*". In *Empirical Studies of Programmers: Fourth Workshop*, J. Koenemann-Belliveau, T. G. Moher, and S. P. Robertson (eds.), pp. 65–81, Intellect Books, 1991.

[18] Y. Gil and G. Lalouche, "*On the correlation between size and metric validity*". *Empirical Softw. Eng.* **22(5)**, pp. 2585–2611, Oct 2017, DOI: 10.1007/s10664-017-9513-5.

[19] M. Halstead, *Elements of Software Science*. Elsevier Science Inc., 1977.

[20] W. Harrison, K. Magel, R. Kluczny, and A. DeKock, "*Applying software complexity metrics to program maintenance*". *Computer* **15(9)**, pp. 65–79, Sep 1982, DOI: 10.1109/MC.1982.1654138.

[21] S. Henry and D. Kafura, "*Software structure metrics based on information flow*". *IEEE Trans. Softw. Eng.* **SE-7(5)**, pp. 510–518, Sep 1981, DOI: 10.1109/TSE.1981.231113.

[22] F. Hermans, *The Programmer's Brain: What Every Programmer Needs to Know About Cognition*. Manning, 2021.

[23] I. Herraiz and A. E. Hassan, "*Beyond lines of code: Do we need more complexity metrics?*" In *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson (eds.), pp. 125–141, O'Reilly Media Inc., 2011.

[24] A. Jbara and D. G. Feitelson, "*On the effect of code regularity on comprehension*". In 22nd *Intl. Conf. Program Comprehension*, pp. 189–200, Jun 2014, DOI: 10.1145/2597008.2597140.

[25] A. Jbara and D. G. Feitelson, "*How programmers read regular code: A controlled experiment using eye tracking*". *Empirical Softw. Eng.* **22(3)**, pp. 1440–1477, Jun 2017, DOI: 10.1007/s10664-016-9477-x.

[26] D. Landman, A. Serebrenik, and J. Vinju, "*Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods*". In *Intl. Conf. Softw. Maintenance & Evolution*, Sep 2014, DOI: 10.1109/ICSME.2014.44.

[27] O. Levy and D. G. Feitelson, "*Understanding large-scale software systems — structure and flows*". *Empirical Softw. Eng.* **26(3)**, art. 48, May 2021, DOI: 10.1007/s10664-021-09938-8.

[28] T. McCabe, "*A complexity measure*". *IEEE Trans. Softw. Eng.* **SE-2(4)**, pp. 308–320, Dec 1976, DOI: 10.1109/TSE.1976.233837.

[29] A. Meneely, B. Smith, and L. Williams, "*Validating software metrics: A spectrum of philosophies*". *ACM Trans. Softw. Eng. & Methodology* **21(4)**, art. 24, Nov 2012, DOI: 10.1145/2377656.2377661.

[30] T. Menzies, J. Greenwald, and A. Frank, "*Data mining code attributes to learn defect predictors*". *IEEE Trans. Softw. Eng.* **33(1)**, pp. 2–13, Jan 2007, DOI: 10.1109/TSE.2007.256941.

[31] R. Minelli, A. Mocci, and M. Lanza, "*I know what you did last summer: An investigation of how developers spend their time*". In 23rd *Intl. Conf. Program Comprehension*, pp. 25–35, May 2015, DOI: 10.1109/ICPC.2015.12.

[32] M. Muñoz Barón, M. Wyrich, and S. Wagner, "*An empirical validation of cognitive complexity as a measure of source code understandability*". In 14th *Intl. Symp. Empirical Softw. Eng. & Measurement*, art. 5, Oct 2020, DOI: 10.1145/3382494.3410636.

[33] N. Nagappan, T. Ball, and A. Zeller, "*Mining metrics to predict component failures*". In 28th *Intl. Conf. Softw. Eng.*, pp. 452–461, May 2006, DOI: 10.1145/1134285.1134349.

[34] J. Pantiuchina, M. Lanza, and G. Bavota, "*The (mis)perception of quality metrics*". In *Intl. Conf. Softw. Maintenance & Evolution*, pp. 80–91, Sep 2018, DOI: 10.1109/ICSME.2018.00017.

[35] P. Piwowarski, "*A nesting level complexity measure*". *SIGPLAN Notices* **17(9)**, pp. 44–50, Sep 1982, DOI: 10.1145/947955.947960.

[36] C. Politowski, F. Khomh, S. Romano, G. Scanniello, F. Petrillo, Y.-G. Guéhéneuc, and A. Maiga, "*A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension*". *Inf. & Softw. Tech.* **122**, art. 106278, June 2020, DOI: 10.1016/j.infsof.2020.106278.

[37] F. Rahman and P. Devanbu, "*How, and why, process metrics are better*". In 35th *Intl. Conf. Softw. Eng.*, pp. 432–441, May 2013, DOI: 10.1109/ICSE.2013.6606589.

[38] F. Salviulo and G. Scanniello, "*Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals*". In 18th *Intl. Conf. Evaluation & Assessment in Softw. Eng.*, art. 48, May 2014, DOI: 10.1145/2601248.2601251.

[39] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vśquez, D. Poshyvanyk, and R. Oliveto, "*Automatically assessing code understandability*". *IEEE Trans. Softw. Eng.* **47(3)**, pp. 595–613, Mar 2021, DOI: 10.1109/TSE.2019.2901468.

[40] N. Schneidewind and M. Hinchey, "*A complexity reliability model*". In 20th *Intl. Symp. Software Reliability Eng.*, pp. 1–10, Nov 2009, DOI: 10.1109/ISSRE.2009.10.

[41] N. F. Schneidewind, "*Methodology for validating software metrics*". *IEEE Trans. Softw. Eng.* **18(5)**, pp. 410–422, May 1992, DOI: 10.1109/32.135774.

[42] J. Shao and Y. Wang, "*A new measure of software complexity based on cognitive weights*". *Canadian J. Elect. Comput. Eng.* **28(2)**, pp. 69–74, Apr 2003, DOI: 10.1109/CJECE.2003.1532511.

[43] T. Sharma and D. Spinellis, "*A survey of code smells*". *J. Syst. & Softw.* **138**, pp. 158–173, Apr 2018, DOI: 10.1016/j.jss.2017.12.034.

[44] M. Shepperd and D. Ince, *Derivation and Validation of Software Metrics*. Clarendon Press, 1993.

[45] M. Shepperd and D. C. Ince, "*A critique of three metrics*". *J. Syst. & Softw.* **26(3)**, pp. 197–210, Sep 1994, DOI: 10.1016/0164-1212(94)90011-6.

[46] D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanović, N.-K. Liborg, and A. C. Rekdal, "*A survey of controlled experiments in software engineering*". *IEEE Trans. Softw. Eng.* **31(9)**, pp. 733–753, Sep 2005, DOI: 10.1109/TSE.2005.97.

[47] J. Spolsky, "*The perils of JavaSchools*". www.joelonsoftware.com/2005/12/29/the-perils-of-javaschools-2, 29 Dec 2005.

[48] A. Trockman, K. Cates, M. Mozina, T. Nguyen, C. Kästner, and B. Vasilescu, "*"automatically assessing code understandability" reanalyzed: Combined metrics matter*". In 15th *Working Conf. Mining Softw. Repositories*, pp. 314–318, May 2018, DOI: 10.1145/3196398.3196441.

[49] J. J. Vinju and M. W. Godfrey, "*What does control flow really look like? Eyeballing the cyclomatic complexity metric*". In 12th *IEEE Intl. Working Conf. Source Code Analysis & Manipulation*, Sep 2012.

[50] E. J. Weyuker, "*Evaluating software complexity measures*". *IEEE Trans. Softw. Eng.* **14(9)**, pp. 1357–1365, Sep 1988, DOI: 10.1109/32.6178.

[51] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "*Measuring program comprehension: A large-scale field study with professionals*". *IEEE Trans. Softw. Eng.* **44(10)**, pp. 951–976, Oct 2018, DOI: 10.1109/TSE.2017.2734091.