DOI:10.1145/3546576

Understanding code depends not only on the code but also on the brain.

BY DROR G. FEITELSON

From Code Complexity Metrics to Program Comprehension

CODE IS HARDLY ever developed from scratch. Rather, new code typically needs to integrate with existing code and is dependent upon existing libraries. Two recent studies found that developers spend, on average, 58% and 70% of their time trying to comprehend code but only 5% of their time editing it.[32,51] This implies that reading and understanding code is very important, both as an enabler of development and as a major cost factor during development.

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 down developers and may even cause them to misunderstand it—possibly leading to programming errors. Conversely, simplicity is often extolled as vital for code quality.

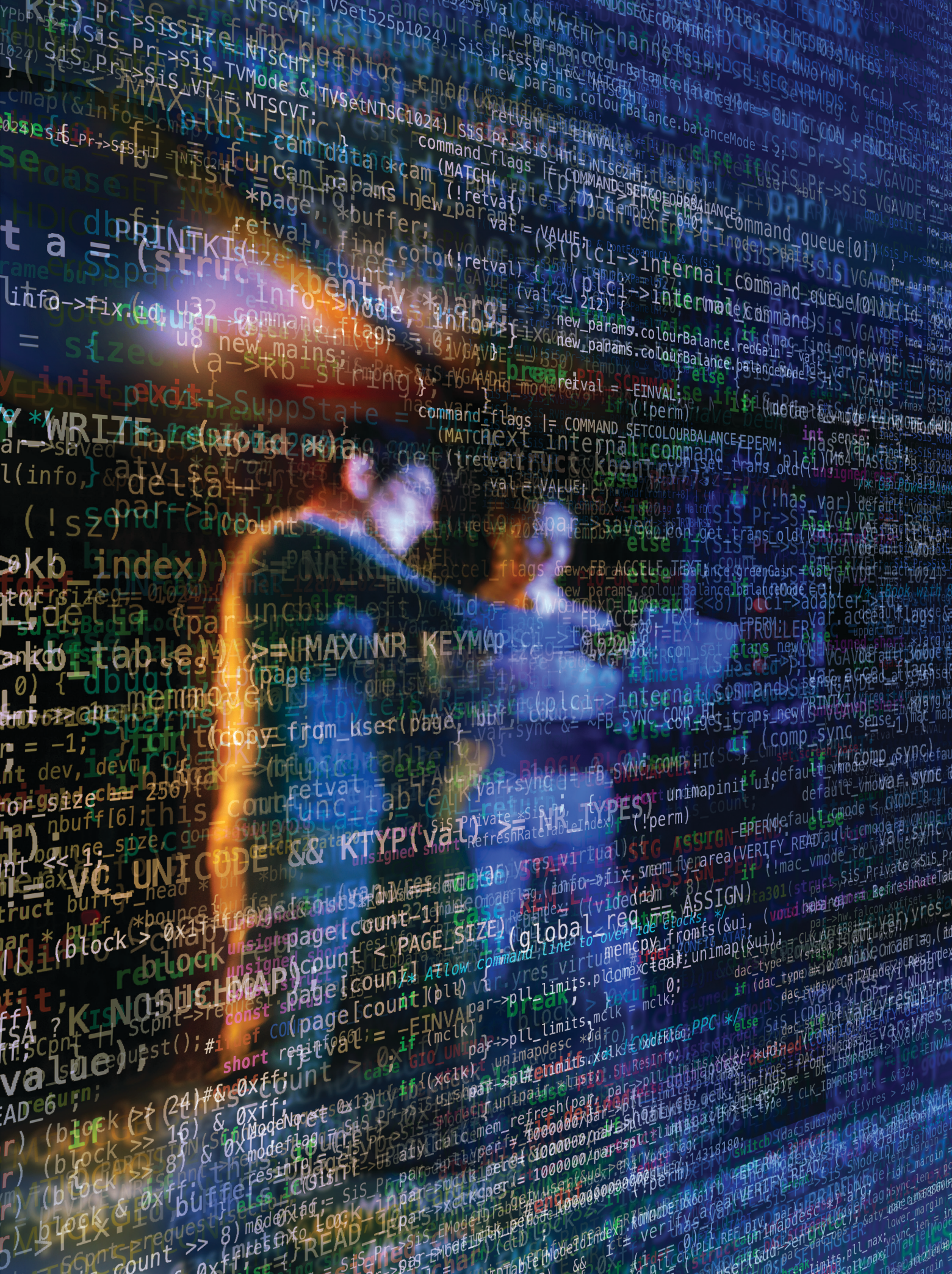
To gain a sound understanding of code complexity and its consequences, we must operationalize this concept. This means we need to devise ways to characterize it, ideally in a quantitative manner. And indeed, many metrics have been suggested for code complexity. Such metrics can then be used for either 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.[16] 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-upon metrics that effectively quantify code complexity. 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

» key insights

- **Code complexity metrics are typically defined based on intuition rather than a systematic empirical process. As a result, they fail to provide an effective quantification of how difficult it is to understand given code.**
- **Careful experimentation can provide data and insights concerning the effect of coding constructs on code complexity and on program comprehension.**
- **The code itself may not be the most important factor. Background and prior assumptions of the developer reading the code also have a significant effect the developer's ability to understand the code.**

IMAGE BY ANDRIJ BORYS ASSOCIATES; PHOTO BY WAVEBREAK MEDIA



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 create more comprehensive metrics. However, as we show, this approach also faces many difficulties, due to the intricacy of the concept of code complexity itself as well as 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.

Metrics for Code Complexity

Various code attributes have been suggested as contributing to code complexity. For example, the use of wild goto statements creates hard-to-follow spaghetti code.[11,36] The use of pointers and recursion is 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.[19] However, understanding code is different from understanding a system as a whole, where the main concern is understanding the architecture.[28] 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 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.[10,17,31] While such correlations have indeed been found,[40] it also appears that no single metric is universally applicable; in fact, all metrics have a poor record of success.[12,14,33,34,39] 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

While literally hundreds of code complexity metrics have been proposed, it has been difficult to find agreed-upon metrics that effectively quantify code complexity.

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,[20] McCabe's cyclomatic complexity,[29] and Henry and Kafura's information flow metric,[22] which were 12 to 19 years old at the time. Were they to write the book today, 30 years later, they would probably have chosen the same three metrics as the most influential (perhaps adding Chidamber and Kemerer's metric suite,[9] which specifically targets 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.[a] 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."[9] 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 an MCC above 10 may need to be simplified, lest they be hard to test.[29] 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.[19,24,45] But others claim that MCC is useful and explain 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.[27]

Other common objections concern

a McCabe's 1976 paper, which introduced MCC,[29] had 8,806 citations on Google Scholar as of March 22, 2023 (304 in 2022, probably not the final count; 352 in 2021; 393 in 2020; 457 in 2019; 433 in 2018, and so on).

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 considered.[21,35] Vinju and Godfrey suggest adding elements such as exit points (`break, continue`) and exception handling (`try, throw`) to the complexity calculation.[49] Campbell went even further, creating a much more comprehensive catalog of constructs, including nesting, `try-catch` blocks, and recursion, which should all be counted.[8]

Given the zoo of constructs that may affect code complexity, another issue is the weight one assigns to different constructs. Intuitively, it does not seem right to give the same level of importance to a `while` loop and a `case` statement. There has been surprisingly little discussion around this issue. Campbell, for example, does not count individual cases at all and gives added weight to each level of nesting. Shao and Wang suggest that if a sequence has weight 1, an `if` should have weight 2, a loop should have weight 3, and parallel execution should have 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.[30]

The common methodology for assessing the utility of complexity metrics is based on correlations with factors of interest.[41] In an extensive recent study, Scalabrino et al. showed 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, Barón et al. have shown that Campbell's metric has a positive correlation with the time needed to understand code snippets.[3] 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.[7]

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?

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.[23] A common approach for research on people's cognitive processes is controlled experiments. In natural sciences fields, 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 requires some code to be understood. Example tasks include determining 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 solution's accuracy, we get some indication of the difficulty.[15] And if the difference between the experimental conditions was limited to some specific attribute of the code, we 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 due in part to the 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 such as list comprehension, which are problematic for developers who are used to more explicit languages. Another common issue to consider is whether students can be used.[4,13] 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 isolated examples of what can be done; they mainly illustrate how much we still need to learn. At the same time, they also indicate that perhaps 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.

Study 1: Controlled Experiments on Loops

The first example is a study motivated by the previously alluded to questions about MCC, 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.

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

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 were meaningful, all the code snippets had the exact same functionality: to determine whether 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. It is also possible to create one large compound conditional which contains a disjunction of conjunctions for the endpoints of each range. One can also use a 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.

Figure 1. Example code snippets use different constructs to express the same functionality. From the top: nested `if`s, a compound conditional, a loop on array, and a loop with arithmetic on loop index.

```
string result = "no...";
if (x >= 10) {
    if (x <= 20) {
        print("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!";
    }
}
```

Code complexity is only one of many factors affecting code comprehension, and perhaps not the major one.

Table 2. Results of comparison of loops with `if`s.

Code Snippet	Time (s)	Error Rate
Nested `if`s	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	--

The results obtained in the experiment for these four options are shown in Table 2; in the full experiment additional structures were also studied—for example, 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—approximately by a factor of 2. This indicates the methodology works: we can design experiments which uncover differences in the 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 `if`s is a reasonable initial estimate, more so than giving them equal weight. However, this cannot be considered the final say on the matter. Many additional experiments with different loop structures and `if`s are needed.

In addition to comparing different constructs, the experiment also included seven 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 six 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, 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 17s to understand, and around 10% of respondents get it wrong. The next three versions each have a simple variation. 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 loops 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 too takes approximately the same time, but now more than half of the respondents make mistakes. The conclusion drawn is that many experimental subjects apparently did not 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. 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—they count down instead of up. This increases the time to correct answer by about 4s, or 25%, which can be interpreted as reflecting extra cognitive effort to figure out what is going on. It also increases the error rate by around 10% relative to the equivalent up-counting loops. These results provide yet another example of the failings of metrics such as MCC: in MCC, loops counting up and loops counting down are equivalent, but for humans they are not.

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 that uses eye tracking to address these issues. Study participants were again given code that they needed to understand, but as they read it, we observed how they divided their attention across repetitions in the code.[26] 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 eight 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, 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.

Table 4. Characteristics of Study 2.[26]

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

Because of the repetitions, regular code tends to be longer and have a higher MCC. For example, in the previously cited image-processing task, 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.[25] This was verified both by metrics such as time-

Figure 2. The results of an experiment comparing the understanding of variations on the canonical `for` loop.

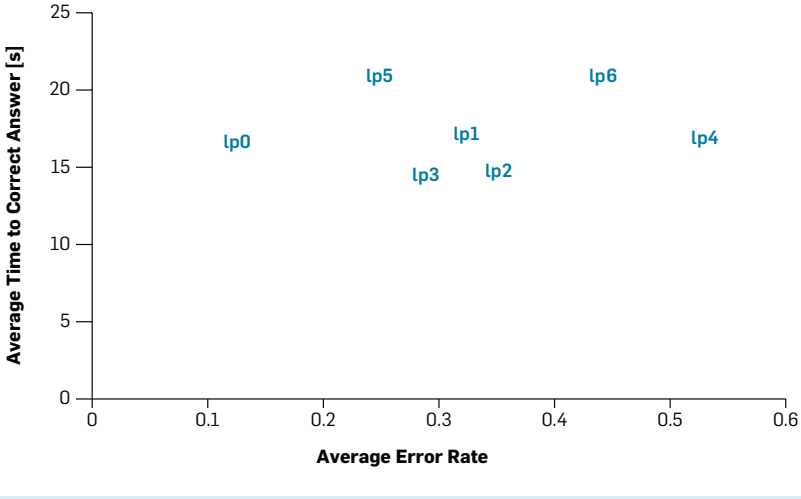


Figure 3. A heatmap of visual attention on regular code, with repeated structures marked (left), and on non-regular code (right).

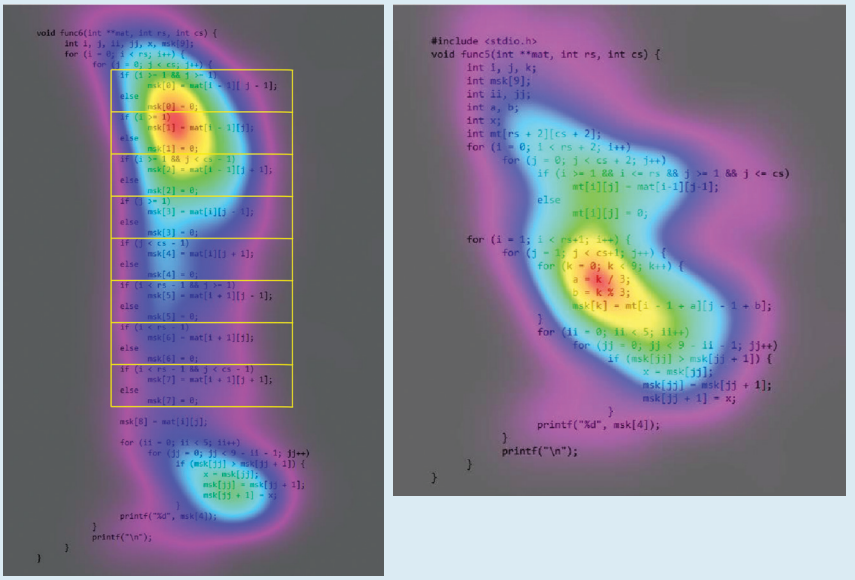
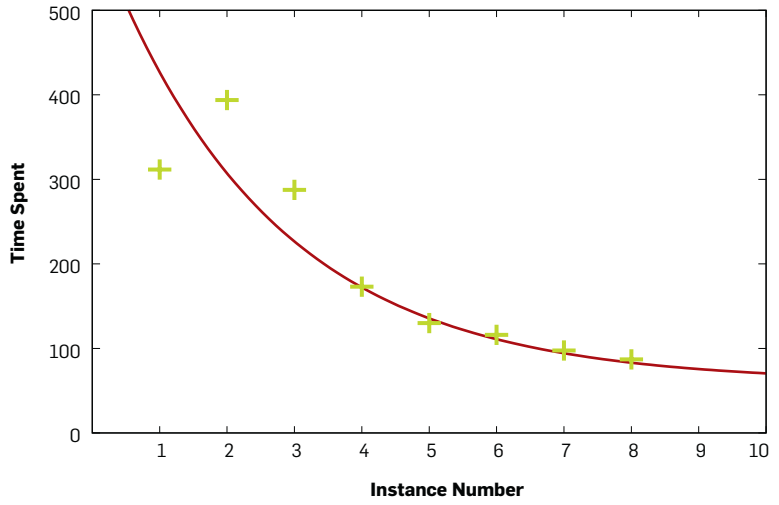


Figure 4. Measurements and model of attention given to structural repetitions in regular code.



to-correct-answer and by subjective ratings. Using a five-point scale from "very easy" to "very hard", subjects rated the regular version as being "easy", "moderate", or "hard" while 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, our eyes do not continuously move. Rather, they fixate on certain points for subsecond intervals and jump from one point of fixation to the next. Cognitive processes, such as understanding, occur during these fixations. So, we can check our prediction by using an eye tracker to measure the number of fixations and total duration on different parts of the code. The heatmap in Figure 3 shows where subjects spent their time. Obviously, in the regular version they spent much more time on the initial repetitions but hardly any time on the later ones. Then, they return to pay attention to the final loops that calculate the median. With the non-regular versions, most attention is focused on the nested loops that scan the image and collect data from the neighboring pixels.

The figure also indicates the "areas of interest" we defined comprising the eight repetitions of the regular structure. Once defined, we can sum up 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.

Study 3: The Effect of Variable Names

A basic problem in understanding code is identifying the domain. For example, are the strings manipulated by this function the names of people? Or 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 mean-

ingful names." Names are therefore extremely important for code comprehension.[5,18,38] The names enable us to understand what the code is about and 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 and the other 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 that the names reflect real practice. We therefore scanned approximately 200 classes from 30 popular (with at least 10,000 stars) Java utility packages on GitHub. Utility packages were used to avoid domain knowledge issues; it was assumed that any developer can understand functions that perform things such as 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 six 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 the study 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 in one-on-one sessions where they explained their thoughts about what they were trying to achieve. If they did not reach a conclusion within 10min, 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, would aid comprehension.

Two examples of the results are shown in Figure 5. These are cumulative distribution functions (CDFs) of the time until 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 graph on the left, 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 2–7min to give an answer. Those in the treatment group, where the variables were renamed to `a`, `b`, `c`, and so forth, took 12–22min. This is 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 six used in the experiment.

But the results of the second example function, `abbreviateMiddle`, are different. In this case, the distributions of times to answers overlap: 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 overlapped, and in one there were another two mistakes, again both by subjects who had received the full-name version.

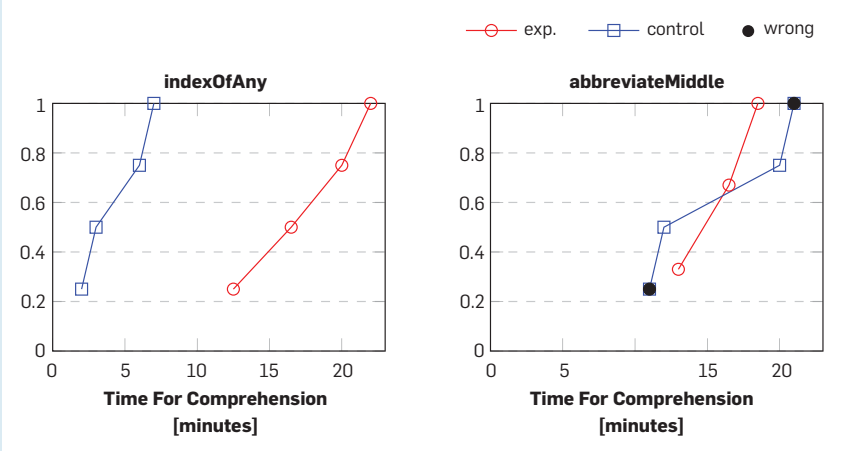
The conclusion from this experiment is that names do not necessarily help comprehension. Moreover, it appears that there are situations where names are misleading, to the degree that they are worse than having meaningless names such as consecutive letters of the alphabet. How can this be? In the few 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 confuse them, because the names do not contain any distinguishing information.

Interestingly, type names can also cause problems. One of the examples was a function that 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

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.



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. 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.

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.[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."[6] 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 must 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 limit the aspiration to analyze code and deduce its objective comprehensibility.

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.

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 here provide examples of such work. Study 1 shows how individual structures can be studied in a quantitative manner. Study 2 concerns ways to combine such results and undermines the idea of just summing the counts of constructs as is commonly done. Study 3 takes 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 consider 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 the real world.

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 in this article 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. Ajami, S., Woodbridge, Y., and Feitelson, D.G. Syntax, predicates, idioms—What really affects code complexity? *Empirical Software Engineering 24*, 1 (February 2019), 287–328; DOI: 10.1007/s10664-018-9628-3.
2. Avidan, E. and Feitelson, D.G. Effects of variable names on comprehension: An empirical study. In *25th Intl. Conf. on Program Comprehension* (May 2017), 55–65; DOI: 10.1109/ICPC.2017.27.
3. Barón, M.M., Wyrich, M., and Wagner, S. An empirical validation of cognitive complexity as a measure of source code understandability. In *14th Intl. Symp. on Empirical Software Engineering and Measurement*, 5 (October 2020); DOI: 10.1145/3382494.3410636.
4. Basili, V.R. and Zelkowitz, M.V. Empirical studies to build a science of computer science. *Communications of the ACM 50*, 11 (November 2007), 33–37; DOI: 10.1145/1297797.1297819.
5. Blinman, S. and Cockburn, A. Program comprehension: Investigating the effects of naming style and documentation. In 6th *Australasian User Interface Conf.* (January 2005), 73–78.
6. Brooks, Jr., F.P., *The Mythical Man-Month: Essays on Software Engineering.* Addison-Wesley (1975).
7. Buse, R.P.L. and Weimer, W.R. Learning a metric for code readability. *IEEE Transactions on Software Engineering 36*, 4 (2010), 546–558; DOI: 10.1109/TSE.2009.70.
8. Campbell, A. Cognitive complexity: A new way of measuring understandability. SonarSource (2016); https://www.sonarsource.com/docs/CognitiveComplexity.pdf.
9. Chidamber, S.R. and Kemerer, C.F. A metric suite for object oriented design. *IEEE Transactions on Software Engineering 20*, 6 (June 1994), 476–493; DOI: 10.1109/32.295895.
10. Cotroneo, D., Pietrantuono, R., and Russo, S. Testing techniques selection based on ODC fault types and software metrics. *J. of Systems and Software 86*, 6 (June 2013), 1613–1637; DOI: 10.1016/j.jss.2013.02.020.
11. Dijkstra, E.W. Go To statement considered harmful. *Communications of the ACM 11*, 3 (March 1968), 147–148; DOI: 10.1145/362929.362947.
12. Fakhoury, S., Roy, D., Hassan, S.A., and Arnaoudova, V. Improving source code readability: Theory and practice. In *27th Intl. Conf. on Program Comprehension*, (May 2019), 2–12; DOI: 10.1109/ICPC.2019.00014.
13. Falessi, D. et al. Empirical software engineering experts on the use of students and professionals in experiments. *Empirical Software Engineering 23*, 1 (February 2018), 452–489; DOI: 10.1007/s10664-017-9523-3.
14. Feigenspan, J. et al. Exploring software measures to assess program comprehension. In *Intl. Symp. on Empirical Software Engineering and Measurement* (September 2011), 127–136; DOI: 10.1109/ESEM.2011.21.
15. Feitelson, D.G. Considerations and pitfalls for reducing threats to the validity of controlled experiments on code comprehension. *Empirical Software Engineering 27*, 6 (November 2022), Article 123; DOI: 10.1007/s10664-022-10160-3.
16. Fenton, N.E. and Neil, M. Software metrics: Successes, failures and new directions. *J. on Systems and Software 47*, 2–3 (July 1999), 149–157; DOI: 10.1016/S0164-1212(99)00035-7.
17. Fenton, N.E. and Neil, M. A critique of software defect prediction models. *IEEE Transactions on Software Engineering 25*, 5 (1999), 675–689; DOI: 10.1109/32.815326.
18. Gellenbeck, E.M. and Cook, C.R. An investigation of procedure and variable names as beacons during program comprehension. In *Empirical Studies of Programmers: 4th Workshop*, J. Koenemann-Belliveau, T.G. Moher, and S.P. Robertson (eds.), Intellect Books (1991), 65–81.
19. Gil, Y. and Lalouche, G. On the correlation between size and metric validity. *Empirical Software Engineering 22*, 5 (October 2017), 2585–2611; DOI: 10.1007/s10664-017-9513-5.
20. Halstead, M. *Elements of Software Science.* Elsevier Science Inc. (1977).
21. Harrison, W. et al. Applying software complexity metrics to program maintenance. *Computer 15*, 9 (September 1982), 65–79; DOI: 10.1109/MC.1982.1654138.
22. Henry, S. and Kafura, D. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering 7*, 5 (September 1981), 510–518, DOI: 10.1109/TSE.1981.231113.
23. Hermans, F. *The Programmer's Brain: What Every Programmer Needs to Know About Cognition.* Manning (2021).
24. Herraiz, I. and Hassan, A.E. Beyond lines of code: Do we need more complexity metrics? In *Making Software: What Really Works, and Why We Believe It*, O'Reilly Media Inc. (2011), A. Oram and G. Wilson (eds.), 125–141.
25. Jbara, A. and Feitelson, D.G. On the effect of code regularity on comprehension. In *22nd Intern. Conf. on Program Comprehension* (June 2014), 189–200; DOI: 10.1145/2597008.2597140.
26. Jbara, A. and Feitelson, D.G. How programmers read regular code: A controlled experiment using eye tracking. *Empirical Software Engineering 22*, 3 (June 2017), 1440–1477; DOI: 10.1007/s10664-016-9477-x.
27. Landman, D., Serebrenik, A., and Vinju, J. Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods. In *Intern Conf. on Software Maintenance and Evolution* (September 2014); DOI: 10.1109/ICSME.2014.44.
28. Levy, O and Feitelson, D.G. Understanding large-scale software systems—Structure and flows. *Empirical Software Engineering 26*, 3 (May 2021); DOI: 10.1007/s10664-021-09938-8.
29. McCabe, T. A complexity measure. *IEEE Transactions on Software Engineering 2*, 4 (December 1976), 308–320; DOI: 10.1109/TSE.1976.233837.
30. Meneely, A., Smith, B., and Williams, L. Validating software metrics: A spectrum of philosophies. *ACM Transactions on Software Engineering and Methodology 21*, 4 (November 2012); DOI: 10.1145/2377656.2377661.
31. Menzies, T., Greenwald, J., and Frank, A. Data mining code attributes to learn defect predictors. *IEEE Transactions on Software Engineering 33*, 1 (January 2007), 2–13; DOI: 10.1109/TSE.2007.256941.
32. Minelli, R., Mocci, A., and Lanza, M. I know what you did last summer: An investigation of how developers spend their time. *23rd Intern. Conf. on Program Comprehension*, (May 2015), 25–35; DOI: 10.1109/ICPC.2015.12.
33. Nagappan, N., Ball, T., and Zeller, A. Mining metrics to predict component failures. In *28th Intern. Conf. on Software Engineering* (May 2006), 452–461; DOI: 10.1145/1134285.1134349.
34. Pantiuchina, J., Lanza, M., and Bavota, G. The (mis) perception of quality metrics. In *Intern. Conf. on Software Maintenance and Evolution* (September 2018), 80–91; DOI: 10.1109/ICSME.2018.00017.
35. Piwowarski, P. A nesting level complexity measure. *SIGPLAN Notices 17*, 9 (September 1982), 44–50; DOI: 10.1145/947955.947960.
36. Politowski, C. et al. A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension. *Information and Software Technology 122* (June 2020); DOI: 10.1016/j.infsof.2020.106278.
37. Rahman, F. and Devanbu, P. How, and why, process metrics are better. In *35th Intern. Conf. on Software Engineering* (May 2013), 432–441; DOI: 10.1109/ICSE.2013.6606589.
38. Salviulo, F. and Scanniello, G. Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically informed study with students and professionals. In *18th Intern. Conf. Evaluation and Assessment in Software Engineering* (May 2014); DOI: 10.1145/2601248.2601251.
39. Scalabrino, S. et al. Automatically assessing code understandability. *IEEE Transactions on Software Engineering 47*, 3 (March 2021), 595–613; DOI: 10.1109/TSE.2019.2901468.
40. Schneidewind, N. and Hinchey, M. A complexity reliability model. In *20th Intern. Symp. on Software Reliability Engineering* (November 2009), 1–10; DOI: 10.1109/ISSRE.2009.10.
41. Schneidewind, N.F. Methodology for validating software metrics. *IEEE Transactions on Software Engineering 18*, 5 (May 1992), 410–422; DOI: 10.1109/32.135774.
42. Shao, J. and Wang, Y. A new measure of software complexity based on cognitive weights. *Canadian J. of Electrical and Computer Engineering 28*, 2 (April 2003), 69–74; DOI: 10.1109/CJECE.2003.1532511.
43. Sharma, T. and Spinellis, D. A survey of code smells. *J. of Systems and Software 138* (April 2018), 158–173; DOI: 10.1016/j.jss.2017.12.034.
44. Shepperd, M. and Ince., D. *Derivation and Validation of Software Metrics.* Clarendon Press (1993).
45. Shepperd, M. and Ince, D.C. A critique of three metrics. *J. on Systems and Software 2*6, 3 (September 1994), 197–210; DOI: 10.1016/0164-1212(94)90011-6.
46. Sjøberg, D.I.K. et al. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering 31*, 9 (September 2005), 733–753; DOI: 10.1109/TSE.2005.97.
47. Spolsky, J. The perils of JavaSchools. *Joel on Software* (December 29, 2005); http://bit.ly/3DPodmh.
48. Trockman, A. "Automatically assessing code understandability" reanalyzed: Combined metrics matter. In *15th Working Conf. on Mining Software Repositories* (May 2018), 314–318; DOI: 10.1145/3196398.3196441.
49. Vinju, J.J. and Godfrey, M.W. What does control flow really look like? Eyeballing the cyclomatic complexity metric. In *12th IEEE Intl. Working Conf. Source Code Analysis & Manipulation* (September 2012).
50. Weyuker, E.J. Evaluating software complexity measures. *IEEE Transactions on Software Engineering 14*, 9 (September 1988), 1357–1365; DOI: 10.1109/32.6178.
51. Xia, X. et al. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering 44*, 10 (October 2018), 951–976; DOI: 10.1109/TSE.2017.2734091.

Dror G. Feitelson (feit@cs.huji.ac.il) is a professor of computer science at The Hebrew University, Jerusalem, Israel.

Watch the author discuss this work in the exclusive *Communications* video. https://cacm.acm.org/videos/understanding-program-comprehension