Syntax, Predicates, Idioms — What Really Affects Code Complexity?

Shulamyt Ajami Yonatan Woodbridge* Dror G. Feitelson
Department of Computer Science and *Department of Statistics
The Hebrew University, 91904 Jerusalem, Israel

***Abstract*—Program comprehension concerns the ability to understand code written by others. But not all code is the same. We use an experimental platform fashioned as an online game-like environment to measure how quickly and accurately 222 professional programmers can interpret code snippets with similar functionality but different structures. The results indicate, inter alia, that `for` loops are significantly harder than `ifs`, that some but not all negations make a predicate harder, and that loops counting down are slightly harder than loops counting up. This demonstrates how the effect of syntactic structures, different ways to express predicates, and the use of known idioms can be measured empirically, and that syntactic structures are not necessarily the most important factor. By amassing many more empirical results like these it may be possible to derive better code complexity metrics than we have today.**

***Index Terms*—Code complexity; program understanding; gamification;**

I. Introduction

Program comprehension is all about bridging gaps of knowledge [6]. Developers often need to understand code written by others. But doing so is notoriously hard and time consuming. The attribute of the code that makes it hard to understand is sometimes called "code complexity". This is an ill-defined term, and people use it in different ways. Factors that may influence complexity include length (more code is harder to understand), syntactical elements (`goto`s are harder than structured loops), data flow patterns (linear is simpler), variable names (which should convey meaning), the way the code is laid out (is it readable?), and more [14], [19], [30], [12], [7], [8], [10], [20].

Once factors influencing complexity are identified, one can try to formulate metrics that quantify the complexity. One of the oldest examples is MCC (McCabe's Cyclomatic Complexity), which essentially counts branch points in the code [29]. This was meant mainly as a testing-complexity metric, meaning it provides a lower bound on the number of tests required to exercise all paths in the code. However, it is often used as a metric for general conceptual complexity, partly because there are no widely agreed alternatives. At the same time, using MCC to measure complexity has met with heated debate [39], [44], [17], [4], [24].

MCC gives the same weight to all control structures. But intuitively a `while` loop feels more complicated than an `if`, a `case` in a `switch` is less complicated, nested `ifs` feel more complicated than a sequence of `ifs` [34], and so on. However, proposals to give different constructs different weights did not report empirical evidence supporting the proposed weights [38], [18]. We wanted to *measure* how different constructs affect understanding, thereby quantifying their contribution to complexity.

Following the tradition of "micro-benchmarks" used in performance evaluation, we started by writing short code snippets that isolate various basic structures, and used them in controlled experiments to measure their effect. But this led to two problems. First, there are endless possibilities, which create the danger of confounding factors that will prevent meaningful analysis. Second, based on experience with various code fragments, we felt that some of the most important factors may not be related to differences between basic constructs, but rather to the composition of conditionals and to using or violating common programming idioms.

We therefore made the following decisions. First, we focused on one specific well-defined family of code snippets: checking whether a number is in any of a set of ranges. This can be expressed in a wide variety of ways, using different syntactical constructs and conditionals, thereby enabling meaningful comparisons. Second, we extended the scope to add compound conditional expressions. Third, we included some specific idioms and their violations that may have an effect on understanding despite being syntactically similar (e.g. a loop counting up vs. a loop counting down).

The following sections present the experimental design in detail, followed by results of experiments with 222 programmers. They embody the following contributions:

- Development of an experimental platform where subjects participate in an online game with the objective to correctly interpret code snippets.
- Generation of an initial set of snippets that allow comparisons between syntactic and other differences.
- Empirical quantification of differences between constructs (e.g. `for` loops are significantly harder than `ifs`), predicates (some but not all negations make predicates slightly harder), and idioms (loops counting down are slightly harder than loops counting up, despite being syntactically identical).

II. Related Work

Surprisingly, there has been relatively little empirical work on how program structures affect comprehension. For example, McCabe famously conjectured that the cyclomatic number of a function's control flow graph reflects its complexity, but did not put this to the test with any real programmers [29]. He also suggested an extended version where the components of compound conditionals are enumerated separately, which in a sense anticipates our work. This was later discussed by Myers, but again with no empirical grounding, instead saying that "it is hoped that the reader will intuitively arrive at the same conclusion" [31]. Over the years there have been some reports that this and related metrics can be used to predict code quality [11], [30], [33], [37], but also reports that claimed that any correlations with these metrics are low or nonexistent [12], [15], [25].

Mynatt considered how the use of iteration vs. recursion affects comprehension, as measured by being able to recall programs [32]. Iselin studied looping constructs and the interplay between writing positive/negative conditions and whether they evaluate to true or false, in an effort to substantiate theories of the cognitive processes involved in comprehension [22]. We focus more on how (compound) conditionals are expressed, and on quantifying this effect. Other studies have also aimed at theories of cognitive processes [40], [6], [27], [43], [3], [35]. In our opinion much more experimental data is needed for such theorizing.

Another structural element of programs that may affect comprehension is successive repetitions. The idea is that repetitive code is easier to understand, because once one understands a certain code fragment, this understanding can be leveraged for its repetitions [42], [24]. At this stage we only consider basic structures in isolation.

The importance of programming idioms (or in their terminology, "plans"), was studied by Soloway and Ehrlich in the 1980s [41], in the context of studying the differences between experts and novices. One of the findings was that experts know of and exploit idioms. The related notion of design patterns was later popularized by the "gang of four" in the context of object-oriented programming [16]. Since then there has been some empirical research on the actual impact of using patterns, but not very much, and the cumulative results are not conclusive [2]. Our work does not deal with these classic design patterns; rather, we focus on more elementary idioms such as a simple loop on an array. Moreover, we study the effect directly on understanding and not on software quality metrics or maintainability as is often done. As far as we know the effect of using (or violating) such basic idioms on comprehension has not been studied before.

III. Research Questions

Our *goal* is to measure how different syntactic and other factors influence code complexity and comprehension. To concretize this goal, in this research we focused on the following *research questions*:

1) What is the effect of control structures on code complexity? More specifically, is the complexity of an `if` the same as that of a `for`?
2) What is the effect of different formulations of conditionals on code complexity? This includes
   a) What is the effect of the size (number of predicates) of a logical expression?
   b) What is the relative complexity of expressing a multi-part decision using a single compound logical expression as opposed to a sequence of elementary ones?
   c) What is the relative complexity of a "flat" formulation and a nested one?
   d) What is the effect of using negation?
3) How does the use (or violation) of programming idioms affect the complexity of the resulting code?

The *metrics* used to investigate these questions are **time** and **correctness** that are measured in a controlled experiment, in which different code snippets with different constructs are presented to programmers. The experimental task is to identify the printout of these snippets. A longer time or more errors are assumed to reflect difficulties in understanding and hence more complex code.

IV. Experimental Design

Our experiment is based on showing experimental subjects short code snippets which they need to interpret. A major issue is what code snippets to use.

A. Considerations for Code Snippet Selection

Since the number of possible code snippets is endless, they need to be chosen carefully taking several considerations into account.

First and obviously, the code snippets need to answer the research questions. We need to include snippets using different constructs, simple or compound conditions, flat/nested structure, with or without negation, and with natural scaling to different sizes.

Second, differences in difficulty need to come *only* from the code's structure. A major gap that may exist between whoever wrote some code and whoever is trying to understand it concerns domain knowledge. For example, if a program embodies certain business rules, a reader who doesn't know these rules will find it difficult to understand. This is unrelated to the programming. If we want to study the difficulties in understanding *code*, we need to factor out any domain knowledge and focus on the code elements. Therefore, the code snippets need to have some well-known common ground, so that the variation between them will not come from their content but from how they express it.

Third, will the code snippets be synthetic or real? Obviously it is better to have an experiment on snippets taken from a real program, but this may cause noise that will cloak the subtle effect we want to measure. Also,

it may be hard to find suitable code snippets that have common ground as we desire. We therefore decided to settle on code written for our experiments.

Finally, a deeper issue is the major question of what exactly we mean by "understanding the code", and how code snippets can be used to assess it at all. In principle one can distinguish between *interpretation* (being able to trace the execution of the code and find its output) and *comprehension* (being able to state the objective of the code). For example, consider the following code snippet:

```
sum = 0;
for (i=1; i<=100; i++)
  sum += i;
```

one can immediately see that it calculates the sum of all numbers from 1 to 100 — what we would classify as comprehension. And if asked about the outcome, we can calculate that it is 5050 without actually running all 100 iterations in our head. But in other cases the functionality is not so transparent, and we do need to mentally execute the code to find its outcome. Thus it might be claimed that resorting to interpretation is a sign of complexity. However, different people may be able to comprehend different snippets easily, so the distinction is probably not universal. We therefore decided not to try to distinguish between different levels of understanding at this stage, and let subjects use whatever works for them to find the outcome of snippets. Investigating the distinction between interpretation and comprehension further is left to future work.

B. Choosing a Common Framework

Taken together, these considerations led to the decision to use a set of synthetic code snippets with common functionality, that can each be employed in the context of multiple research questions. The chosen functionality is to test whether a number is in a collection of non-overlapping ranges. Each range is defined by a conjunction of two simple Boolean atoms with > and < comparisons. We assume that understanding such atoms is easy, and specifically do not test for edge cases to avoid possible confusion. The same atoms are used in all the snippets, so the variation is caused only by the syntactic ways of using and combining them.

In this framework all the snippets have common ground from a very basic domain, they are easy to scale (add more ranges), and the functionality can be expressed in many different ways. For example, consider the question of verifying whether `x` is in either of two successive ranges (a,b) and (c,d), where a<b<c<d. We can express this using a single `if/else` statement with a compound logic expression:

```
if (x>a && x<b || x>c && x<d) {print(1); /*in*/}
else {print(2); /*out*/}
```

Alternatively, this can be broken into separate `if/else` statements that together achieve the same goal:

```
if (x>c) {
    if (x<d) {print(1); /*in*/}
    else {print(2); /*out*/}
}
else if (x>a) {
    if (x<b) {print(3); /*in*/}
    else {print(4); /*out*/}
}
else {print(5); /*out*/}
```

The snippets include print statements that identify the path in the code. This is done using single-digit numbers rather than strings like "in" and "out" to avoid giving hints regarding the functionality. The outcome of each snippet is one such digit being printed, and this is what the subjects are asked to identify. In the real experiments the "in" and "out" comments are of course excluded.

C. Conversion Rules

In order to be able to compare snippets consistently, we define precise conversion rules for compound predicates in `if`s. Conjunction (AND) in a compound expression converts to an additional `if` nested in the `then` block:

```
if(A && B){print(1)}          if(A){
else{print(2)}        ⟷        if(B){print(1)}
                                 else{print(2)}}
                               else{print(3)}
```

Disjunction (OR) converts to an `elseif` (or equivalently an `if` nested in the `else` block).

```
if(A || B){print(1)}          if(A){print(1)}
else{print(2)}        ⟷       else if(B){print(2)}
                               else{print(3)}
```

Brackets are added only if necessary, meaning that the expression outcome will be different without brackets.

D. Creating the Pool of Snippets

Given the common framework described above, we need to create specific code snippets to use in experiments and answer the different research questions.

We start by defining three main variations of the code structure, with differences in nesting. These reflect RQ 2c:

a. The simplest and most straightforward way is testing whether a number is in the first range or in the second range or in the third range and so on. This is a flat structure.
b. A two-level scheme, first testing whether a number is between a lower and an upper bound, and then whether it is in some range in between.
c. Divide the whole range into two, test whether the number is in the first part or the second part, then continue recursively inside those parts drilling down to the individual ranges. This is the most nested structure.

Each of these variation is then expressed in two main ways: using a single `if/else` statement with a compound logical expression, or using control flow with multiple `if/else`s with a single condition in each one (as in the example above). This reflects RQ 2b. In the sequel, the first three will be denoted al, bl, and cl (l for logic), the later three

TABLE I
CODE SNIPPETS USED FOR EACH RESEARCH QUESTION

RQ	*description*	*snippet comparisons*
1	`if` vs. `for`	as–cs–f*–f[], f*–f[]
2a	expression size	more or fewer ranges
2b	compound vs. structure	as–al, bs–bl, cs–cl
2c	flat vs. nesting	as–bs–cs, al–bl–cl
2d	negation	al–an–an1–an2
3	loop idioms	lp0–lp1–...–lp5–lp6

as, bs, and cs (s for structure). Actual code will be shown below when we discuss it in the results.

The next step was to express the first variant (al above) with negation, in three different variants that will be denoted `an`, `an1`, and `an2` (the differences are detailed below). This reflects RQ 2d.

A harder problem is to accommodate different control structures. It initially seems that `if` is intrinsically different from `for`: the first denotes a branch, the second a loop. But when we want to check inclusion in multiple ranges, this can be done either by `if`s as shown above, or by a loop that traverses all the ranges and checks them one at a time. This observation facilitated creating snippets using a `for` loop, used for RQ 1, with two versions:

- By setting the ends of the ranges to be multiples of 10, say 0 to 10, 20 to 30, and so on, we can express them by simple arithmetic manipulations of the `for` loop iteration variable (variant f*).
- Alternatively it is possible to store the ranges' end points in an array and to go over them in a sequential manner (variant f[]).

These are both reasonable uses of `for` loops.

All these code snippets were then scaled with different numbers of ranges: 2, 3, and 4, for RQ 2a.

Finally, we need snippets which reflect idioms and their violation, for RQ 3. We decided to use idioms of `for` loops. These are denoted lp0 to lp6, and included the following:

- Loops with different end conditions, comparing the classic loop `for (i=0; i<n; i++)` with variants starting from 1 and/or using `<=` as the condition. An example is `for (i=1; i<n; i++)`, which does not cover the full conventional range.
- Comparing a loop counting up with the same loop counting down, e.g. `for (i=n-1; i>=0; i--)`.

A summary of the snippets and their relationship to research questions is given in Table I.

E. Generating a Test Plan

All told we have 40 code snippets: 12 each with 3 and 4 ranges, only 9 with 2 ranges (because some cases become identical for only 2 ranges, e.g. al and cl), and 7 special loop cases. In the pilot we found that this is too much for a single subject to perform, not because of the time investment (many snippets take only 10–20 seconds to solve) but because they are repetitive causing reduced focus and a learning effect. So we need to select a subset for each subject.

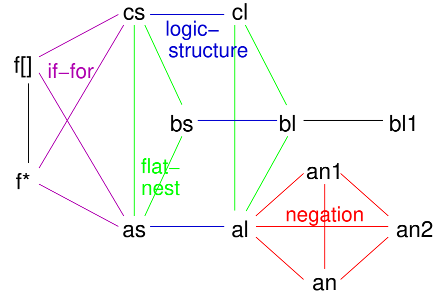

Fig. 1. Code snippet comparisons in relation to research questions.

The selection needs to be done efficiently in terms of including pairs or sets of snippets that are meaningful to compare to each other. For example, snippet al (simple logic expression) can be compared with as (an equivalent structure of `if`s) in the context of RQ 2b. It can also be compared with snippets an, an1, and an2 (alternative expressions using negation) to answer RQ 2d, and with snippets bl and cl (different nesting structure) for RQ 2c. The full graph showing all pairs of comparable snippets (except those relating to size) is given in Figure 1.

Based on the above, the selection of snippets to present to a subject is done as follows.

- We formed three partly overlapping subsets based on the research questions: {as, cs, f*, f[]}, {as, al, bs, bl, cs, cl}, and {al, an, an1, an2}. By selecting all the snippets in such a group, we collect data that facilitates within-subject comparisons of all the relevant pairs of snippets described above. For each subject we pick one of these three groups at random.
- The snippets in the above group are used in their 3-range version. In addition we select two of them at random and add their 2-range and 4-range versions (if they exist — not all snippets have a meaningful 2-range version).
- Next, we add three special idiom snippets drawn randomly.

The selected snippets are presented in a random order. The total number of snippets presented to each subject is between 11 and 14.

V. EXPERIMENTAL PLATFORM

Given a well defined pool of snippets, we need to design the experimental platform to present them to subjects.

A. Considerations and Implementation Principles

Our metrics — from which we deduce the effect of syntactic factors — are the time and accuracy with which subjects interpret the code. But this may be a subtle effect. The snippets are at a basic level and very short. So we need high accuracy in measurement and many samples, meaning many subjects.

Get the CODE?

Fig. 2. Stylized game slogan.

We also need to motivate the subjects to do their best in terms of both time and accuracy. This typically involves a tradeoff: higher accuracy requires more time, so we don't want subjects to spend much time rechecking their work. Moreover, the subjects won't be under our control, and monetary compensation will probably not work to get experienced professionals to participate in such a short experiment. So how we are going to motivate the subjects?

Our solution to these problems comes in two levels:

- Technological: To reach many subjects and achieve accurate measurements, we implement a *website* for the experiment. Participation is then easy (send a link, no installation needed in the client, cross platform so any OS with a browser can run it).
- Methodological: To motivate the subjects we design the website based on some *gamification* principles. Huotari and Hamari define gamification by the user experience, which should be fun and challenging [21]. Deterding et al. say that gamification is reflected in the system implementation, by including game elements like graphics, an avatar, a timer, a progress bar, feedback, etc. [13]. We will show how we implement gamification according to both these definitions.

B. The Platform

We present the system by describing its flow.

1) When a subject visits the experiment website, the landing page is a welcome screen with an explanation of how the experiment will be conducted. It includes statements that the topic is code comprehension, that we are evaluating the code and not the participants, and that they may retire before finishing if they wish. We add gamification elements like a cartoon of a programmer in action that will act as an avatar, so the subject can identify with him through the experiment. We also use a stylized slogan ("Get the code", see Figure 2) to create a game context, and label the 'next' button with 'Got you!' in order to create an enjoyable atmosphere. At this point, the server chooses a test plan randomly according to the procedure described earlier.
2) A popup is opened with a demographic questionnaire with details on education and experience. None of the fields are mandatory. Notice that choosing a test plan does not dependent on experience or any other demographic information of the user.
3) Then an example screen is displayed, showing how the experimental screen looks and pointing out the function of the different parts.
4) Now the actual experiment starts. The main screen is shown in Figure 3. A code snippet is presented in the white window to the left. If the snippet is too long, scrollbars are automatically added. The subject should write the snippet's output in the black window to the right. Additional important elements in this screen include:

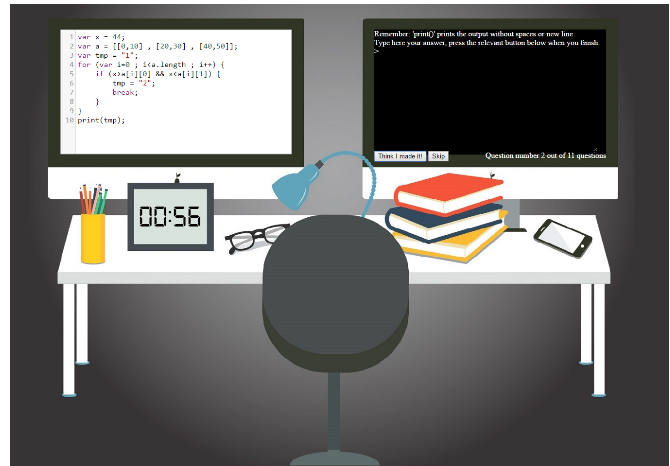

Fig. 3. Screenshot of gamified experimental platform.

- The main screen is decorated as an office from the perspective of a programmer. This graphic is a gamification element that creates atmosphere and context.
- A timer counting down to 0. The timer provides a constant reminder to answer as fast as possible. This is a known gamification element that adds challenge and motivation. In most of the questions the clock counts back 60 seconds. For questions where the pilot indicated subjects had trouble answering within a minute we gave 90 seconds.
- A progress bar, showing the place in the sequence of questions. This is also a gamification element that keeps subject motivated by keeping them aware of their progress.
- Two buttons at the bottom of the reply screen, labeled "I think I made it" and "skip". The first is used if the subject believes he managed to solve the challenge. The second allows him to skip the question; we provide this option to reduce the motivation to guess. When the subject clicks either of these buttons a small popup appears, showing the correct answer and the user's answer, and giving a short compliment if the subject got it right ("Wow", "Nice :)", "Good!"). Such feedback is a gamification element that on the one hand motivate the user, and on the other hand helps to do better next time by learning from mistakes.

  The popup also contains a button labeled "let's continue!" to enable the subject to move on. This allows each subject to advance at his own rate. In particular, subjects that take a part in the experi-

ment remotely may be disturbed and may not do it continuously.

Every time a user completes a question we save his answer and the time it took him asynchronously to the server. Thus, if subjects decide to quit the experiment in the middle, we can still use the results for those snippets they did do. In addition, the correct answer to the question is not loaded together with the question, but only after the answer is submitted, so subjects cannot peek using the browser console.

5) After completing the experiment a goodbye screen with a thank-you message and a summary of the results is shown. We also added a reminder that the scores are not comparable, as each subject gets different snippets and some are harder than others. This was because we noticed in the pilot that a competitive atmosphere was created, and some of the subjects who achieved low results were disappointed.

VI. Experiment Execution

A. Subjects

Experimental subjects were recruited by soliciting their help. These were all professional developers, starting with colleagues of the first author working at the same multinational software company, and then continuing via word of mouth to other departments and companies. Most were from Israel, but some came from locations of the same companies in India and the UK. Sending an email with a deadline ("we need the results of this experiment by Tuesday") led to 180 responses within 3 days, after a previous request without a deadline yielded only 40 in a whole week.

The results presented here are based on 222 subjects who participated through the Internet during June to August 2016. (In addition there were about 30 in the pilot and 25 who were observed personally.) 119 of them were male and 103 female. The average age was 28.9 (range of 21–56). 151 of them had an academic degree: 124 BSc, 17 MSc, and 10 PhD. Levels of experience ranged from 0 to 19 years, with an average of 5.6 years.

B. Variables

The most important *independent variable* is obviously the code snippets. But there are also other independent variables that may cause confounding effects. This includes the demographic variables (level of experience, level of education, sex). Another is the order that the snippets are presented, as the common framework behind the snippets may lead to learning effects. But the effect of these variables is mitigated by randomization.

The main *dependent variables* are correctness and time. Code snippets that are more complex are expected to require more time and lead to more mistakes. Time is measured from displaying the code until the subject presses the button to indicate he is done. In our analysis we focus exclusively on the time for correct answers since incorrect answers may reflect misunderstandings, or guesses, or giving up. Another dependent variable is the button the subject chose to click: either "I think I made it" or "skip". However, skip was used only 27 times in total, out of 2326 recorded answers, so its effect is negligible.

C. Statistical Methodology

1) Within subject design: A major issue in software engineering empirical research is individual differences between experimental subjects. It is commonly thought that such differences can reach a factor of 10 or more [36], [9], [26]. This causes the results to have a large variance, which may mask experimental effects.

A possible solution to this problem is to use within subject comparisons and paired samples. Thus, when we want to compare performance on snippets `a` and `b`, we measure how the *same subjects* perform on both. We then analyze the differences rather than the raw results, which factors out individual differences to a large degree. Our assignment of snippets to subjects facilitates this design.

2) Statistical Approach: We denote $X_{i,j} = 1$ if subject i's answer to code snippet j is correct, and $X_{i,j} = 0$ if wrong. Moreover, we define $Y_{i,j} \in R^+$ to be the time to reach a correct answer, with the same indexes. The matrices $X_{i,j}$ and $Y_{i,j}$ contain many missing values, since subjects are not tested on all snippets.

Suppose we have two vectors of code snippets, such that the differences between them reflect one of the research questions (e.g. one uses negation and the other doesn't). We denote them by $\mathcal{A} = (a_1, .., a_n)$ and $\mathcal{B} = (b_1, .., b_n)$. The terms a_k and b_k are snippets' identifiers. Note that the members of each vector are not necessarily all different, as we may want to compare the same snippet against several others. The snippets are ordered so that a_k and b_k are considered a pair (à la Figure 1). Let P_k denote the index set of all subjects who answered correctly both snippets of the pair a_k and b_k.

We first analyze the time difference of correct answers to $\mathcal{A}$ and $\mathcal{B}$. Denote by t_j the time distribution of a correct answer to code snippet j. Our null hypothesis is that $t_{a_k} = t_{b_k}$ for every $k = 1, ..., n$. In other words, the correct answer distributions of pairs are identical. We apply a non-parametric permutation test, where the test statistic is defined as follows. Let $D_k = (Y_{i,a_k} - Y_{i,b_k})$ for $i \in P_k$ be the time difference vector, for the k'th snippets pair, of all subjects in P_k. Then,

$$T_k = \sqrt{|P_k|}\frac{\overline{D_k}}{\mathrm{SD}(D_k)}, \quad (1)$$

where SD stands for standard deviation. The test statistic T is the mean of all T_k's. In this way we account for the difference within each set. If $|\mathcal{A}| = |\mathcal{B}| = 1$ (we're comparing just one pair of snippets) T becomes the ordinary standardized mean difference.

Because the null hypothesis states identical time distribution of snippet pairs, exchanging observations within

pairs does not change T's distribution. To obtain T's empirical distribution, we calculate 2×10^4 values of T corresponding to random selections of what observations to flip. To test if $t_{a_k} - t_{b_k} > 0$ on average for $k = 1,..,n$ (standardized average as above), we calculate the upper tail region of T. To test if $t_{a_k} - t_{b_k} \neq 0$, using the fact that T is symmetric around zero under the null hypothesis, we consider the distribution of the absolute value statistic, and calculate the upper region of $|T|$.

We also perform a Wilcoxon signed rank test. Denote by $R_{i,k}$ the rank of $|Y_{i,a_k} - Y_{i,b_k}|$. We compute the value $R^{'}_{i,k} = \text{sgn}(Y_{i,a_k} - Y_{i,b_k})R_{i,k}$, and then calculate T_k as in (1), using all $R^{'}_{i,k}$ that belong to the snippet pair (a_k, b_k). Finally, we obtain the mean of T_k, $k = 1, ..., n$. Calculation of significance was done as previously, with 2×10^4 random permutations.

In order to estimate the difficulty of snippets, we use the Rasch model applied to the error rates [1], [5]. Denote the ability of subject i by θ_i, and the difficulty of snippet j by β_j . The essence of the model is to express the probability of a correct answer as a logistic function of the difference between the ability and the difficulty:

$$\Pr(X_{i,j} = 1) = \frac{e^{\theta_i - \beta_j}}{1 + e^{\theta_i - \beta_j}}. \quad (2)$$

As the difficulty parameter β_j increases, the probability of the event $\{X_{i,j} = 1\}$ decreases. Note that the model is not identifiable, since adding a constant c to β_j and θ_i does not change expression (2). We therefore add a restriction that the sum of all estimated βs be 0. To estimate the βs we use the conditional maximum likelihood (CML) approach, which conditions on the number of correct answers of each subject as a sufficient statistic for θ_i.

To compare difficulty between groups $\mathcal{A}$ and $\mathcal{B}$, we consider the contrast statistic

$$C(\mathcal{A}, \mathcal{B}) = \frac{1}{|\mathcal{A}|} \sum_{j \in \mathcal{A}} \hat{\beta}_j - \frac{1}{|\mathcal{B}|} \sum_{j \in \mathcal{B}} \hat{\beta}_j. \quad (3)$$

We can calculate the variance given the covariance matrix Σ of $\hat{\beta}$. In order to obtain the p-value, we assume normality of $\frac{C(\mathcal{A},\mathcal{B})}{\sqrt{\text{Var}(C)}}$, since the number of degrees of freedom is large. All this procedure was done using the eRm package [28].

VII. Results

A. Descriptive Statistics

An example of the raw results is shown in Figure 4. This includes the distributions (CDF) of the time to achieve a correct answer for two snippets: al (basic simple formulation) and an2 (a formulation with negation). The graphs show that, except perhaps in the tails, the distribution for an2 dominates that of al, and indeed the analysis below shows that the difference is statistically significant.

However, it is interesting to also note the distribution of differences between pairs of results by the same subjects. This turns out to include both positive and negative

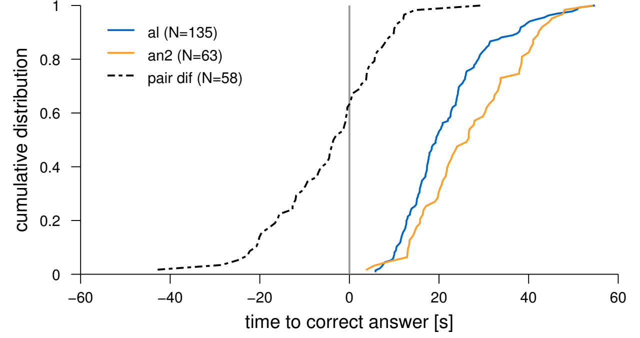

Fig. 4. Distributions of time to correct answer for two snippets.

results that are quite high. This means that despite the general tendency to do better on al, some subjects actually did better (and even significantly better) on an2. This happened in practically all comparisons.

B. Results of Statistical Analysis

Results comparing various groups of snippets are given in Table II. These include p-values from permutation and Wilcoxon tests on the time to correct answers, and from contrasts on the number of wrong answers. Note that they do not always correspond to each other, as it may happen that in a certain comparison there is a difference in times but not in error rates, or vice versa.

When comparing the p-values to a threshold of 0.05, 5% of them by definition should be found to be "statistically significant" even if the null hypothesis holds (type I errors). Therefore a Bonferroni correction (dividing the threshold by the number of tests) is typically used when multiple tests are performed. However, we performed a total of 31 tests, and around half (rather than only 1 or 2) were below the 0.05 threshold, so the majority can be assumed to hold anyway. Most are also below a Bonferroni corrected threshold.

1) RQ 1: `if` *vs.* `for`*:* The results show a significant difference between snippets based on ifs and those using for. The former are represented by snippets as and cs, and use a nested sequence of > or < tests to establish inclusion in a set of ranges, while the latter do it with loops using either of the following styles:

f*:
```
for (var i=0 ; i<3 ; i++)
   if (x>10*2*i && x<10*(2*i+1)) { ... }
```

f[]:
```
var a = [[0,10] , [20,30] , [40,50]];
for (var i=0 ; i<a.length ; i++)
   if (x>a[i][0] && x<a[i][1]) { ... }
```

The snippets using for took more than twice as long to interpret, and led to more errors. All the differences were statistically significant (Table II). This implies that metrics like MCC that assign the same complexity to all branching instructions may be too simplistic. Furthermore, the f* variant took much longer than the f[] variant,

TABLE II
RESULTS OF COMPARISONS RELATED TO EACH RESEARCH QUESTION

		compare this...			*...with this*					*p-value*		
RQ	*description*	*snippet*	*avg±stdv*	*err*	*snippet*	*avg±stdv*	*err*	*N*	*sd*	*permutation*	*Wilcoxon*	*contrast*
1	`if` vs. `for`	f*,f[]	40.5±17.8	0.387	as,cs	17.8±9.8	0.213	79	1	0**	0**	0.0003**
		f*	46.6±19.0	0.325	as	18.3±11.2	0.175	24	1	0**	0**	0.0072*
		f[]	35.0±14.7	0.450	as	15.9±11.1	0.175	19	1	0**	0**	0.0002**
		f*	43.6±19.3	0.325	cs	20.3±9.3	0.250	19	1	0.0001**	0.0001**	0.0813
		f[]	34.5±15.0	0.450	cs	16.3±6.4	0.250	17	1	0.0001**	0.0001**	0.0049*
		f*	45.6±21.9	0.325	f[]	33.2±13.2	0.450	14	2	0.0578	0.0785	0.291
2a	size	3 seg	24.0±13.1	0.206	2 seg	20.6±13.4	0.145	235	1	0**	0**	0.0081*
		4 seg	24.4±12.9	0.333	3 seg	21.5±11.7	0.236	226	1	0.748	0.0727	0.175
2b	expr vs. struct	al,bl,cl	21.2±8.6	0.142	as,bs,cs	19.5±9.6	0.208	115	1	0.0423*	0.0288*	0.970
		al	19.2±8.4	0.082	as	17.1±8.8	0.175	43	1	0.0949	0.0895	0.999
		bl	22.5±9.4	0.182	bs	19.9±9.4	0.190	38	1	0.0630	0.0528	0.535
		cl	22.3±7.7	0.254	cs	22.0±10.3	0.250	34	1	0.429	0.358	0.544
		bl	23.7±11.0	0.182	bl1	21.6±9.0	0.196	35	1	0.133	0.172	0.491
2c	flat vs. nested	as,al	17.5±7.9	0.119	bs,bl	21.1±10.2	0.186	84	2	0.0032*	0.0039*	0.0580
		as	16.6±8.1	0.175	bs	19.9±9.6	0.190	42	2	0.0486*	0.0622	0.961
		al	18.5±7.6	0.082	bl	22.4±10.7	0.182	42	2	0.0237*	0.0192*	0.0097*
		as,al	17.4±9.6	0.119	cs,cl	21.4±9.2	0.252	102	2	0.0002**	0**	0**
		as	16.7±9.9	0.175	cs	20.8±10.1	0.250	63	2	0.0016*	0**	0.175
		al	18.6±9.0	0.082	cl	22.4±7.4	0.254	39	2	0.0192*	0.0140*	0**
		bs,bl	20.1±9.7	0.186	cs,cl	22.2±9.2	0.252	70	2	0.482	0.272	0.120
		bs	20.1±9.7	0.190	cs	22.0±10.6	0.250	38	2	0.339	0.388	0.240
		bl	22.2±9.8	0.182	cl	22.3±7.4	0.254	32	2	0.957	0.495	0.297
2d	negation	an,an1,an2	25.0±10.9	0.232	al	23.5±10.4	0.082	166	1	0.0661	0.148	0.0002**
		an	21.5±8.4	0.162	al	23.5±10.7	0.082	64	1	0.917	0.928	0.0593
		an1	26.7±11.9	0.355	al	24.0±11.0	0.082	44	1	0.0920	0.116	0**
		an2	27.5±11.7	0.182	al	23.0±9.8	0.082	58	1	0.0040*	0.0063*	0.0030*
		an1	28.1±11.9	0.355	an	21.5±8.5	0.162	42	2	0.0008**	0.0004**	0.0020*
		an2	27.6±12.1	0.182	an	21.2±7.1	0.162	55	2	0.0013**	0.0008**	0.273
		an1	26.1±11.7	0.355	an2	26.0±11.9	0.182	41	2	0.973	0.930	0.036*
3	loops	lp2,lp3,lp4	15.0±9.4	0.385	lp0,lp1	15.5±9.0	0.225	103	1	0.831	0.820	0.0003**
		lp5,lp6	20.6±7.9	0.341	lp0,lp1	16.2±8.7	0.225	73	1	0.0002**	0.0003**	0.0215*

0 means $< 10^{-5}$. * denotes statistical significance $p < 0.05$; ** denotes statistical significance also after Bonferroni correction $p < 0.0016$. avg±stdv: of time to correct answer. err: wrong answers rate. N: number of paired samples of correct answers. sd: one-sided/two-sided.

albeit a direct comparison was not statistically significant. This may imply that using arithmetic on the loop index is overly confusing, at least in this case, while referencing successive array cells is more natural.

2) RQ 2a: size of conditional: The size of conditionals is quantified by the number of atomic comparisons they contain. In our code snippets this reflects the number of ranges that are checked. The results were largely as expected. Snippets with 3 ranges took 16.5% more time than snippets with 2 on average, and this was statistically significant. Snippets with 4 ranges also took more time than snippets with 3 on average, but this was not statistically significant. More ranges also led to higher error rates. (Note: in these analyses $N > 222$ because each subject typically had 2 relevant comparisons.)

3) RQ 2b: single expression vs. structure: As noted above in Section IV-C, compound logical expressions composed of many atoms may be converted into a nested structure of simple `if`s. But which of these two structures is easier to handle? The results were that the nested structure took slightly less time on average, but nearly all the comparisons were not statistically significant.

This lack of difference is somewhat surprising, because in debriefings of subjects that were observed during the experiment they reported that the nested structures were significantly easier than the snippets with compound logic expressions. We conjectured that this is because a sequence of `if`s allows one to trace the relevant path through the code for the given input one step at a time, but when faced with a compound expression you need to understand it as a whole.

Note that the lack of significant separation also implies that the structures of many nested `if`s is not significantly harder, even though these code snippets are much longer when counting LOC.

4) RQ 2c: flat vs. nested structure: This distinction is somewhat subtle, and involves the use of a "flat" structure where predicates follow each other on the same level, as opposed to a nested structure where some predicates are subordinate to others. The logical expressions (for 4 ranges) are

```
al: (x>0 && x<10 || x>20 && x<30 ||
              x>40 && x<50 || x>60 && x<70)

bl: (x>0 && x<70 && (x<10 || (x>20 &&
              (x<30 || (x>40 && (x<50 || x>60))))))

cl: (x>40 && (x<50 || x>60 && x<70) ||
              x<30 && (x>20 || x<10 && x>0))
```

Similarly, there were versions with nested `if`s that each

contain a single atom, based on the conversion rules described above in Section IV-C.

As it was not clear which is expected to be easier, we used a two-sided test in this case. The results indicate that the a versions were slightly easier than the other two, and this was statistically significant or nearly so (significance was stronger when comparing with the c versions, which have the deepest nesting).

Note that the bl version has deep skewed nesting, which was required in order to apply the conversion rules. But most programmers would probably avoid this, so we also checked an alternative version:

bl1:
```
(x>0 && x<70 && (x<10 || x>20 && x<30 ||
              x>40 && x<50 || x>60))
```

The results were that there was no significant difference.

5) RQ 2d: use of negation: The logic expressions in the four snippets being compared in this case, for 3 number ranges, are the following:

al:
```
(x>0 && x<10 || x>20 && x<30 || x>40 && x<50)
```

an:
```
(x>0 && x<50 && !(x>10 && x<20) && !(x>30 && x<40))
```

an1:
```
(!(x<0 || (x>10 && x<20) || (x>30 && x<40)|| x>50))
```

an2:
```
(!x<0 && !(x>10 && x<20) && !(x>30 && x<40) && !(x>50))
```

The results were somewhat surprising. Comparing the positive version al to the three negation version, there was a significant difference only when comparing with an2. Moreover, the average time for an was actually a bit *shorter* than for al (although not statistically significantly different). And comparisons of an with an1 and an2 led to strongly statistically significant differences. Thus not every negation causes difficulties to the same degree. Detailed investigation of this effect is left to future work.

6) RQ 3: common loop idioms: A special set of 7 snippets concerned the details of `for` loops on arrays. The specifics are listed in the following table:

version	*init*	*cmp*	*end*	*step*
lp0	0	<	len	++
lp1	0	<=	len-1	++
lp2	0	<	len-1	++
lp3	1	<	len	++
lp4	1	<	len-1	++
lp5	len-1	>=	0	− −
lp6	len-1	>	0	− −

Note that lp2, lp3, lp4, and lp6 do not cover the whole range as may be expected, and are therefore abnormal. The results showed that this did not cause significant differences in processing time, but did lead to significantly more errors. In addition, significant differences were found between loops counting up and loops counting down, which took 27% longer on average.

VIII. Threats to Validity

Several decisions about the experimental design were taken specifically to mitigate threats to validity. However, other threats remain.

Construct validity refers to correctly measuring the dependent variable. In our case this is the time to interpret a certain code snippet, which we relate to its structure. But the way the snippet is written can also have an effect. One problem is that in a flat compound statement the ends of the number ranges can appear in numerical order, but when using nested `if`s the order must be manipulated in correspondence with the structure. For example, in the recursive style it is necessary to start from the middle. Also, different programmers are used to different coding guidelines. Opinions about this sometimes reach religious proportions. For example not placing brackets on a new line may cause annoyance and distraction.

Another issue is the different lengths of the code snippets. Longer code is considered harder to understand, and some say that LOC is the only important metric [20]. In principle we could inflate shorter snippets by adding meaningless lines (e.g. `var z = 1;`), but decided against doing so as this might also confound the results.

Finally, what we really care about is understanding and not time. One can question whether the time and accuracy of providing the output of a code snippet really reflect understanding. We leave the deeper discussion of what exactly is meant by "understanding" and how to measure it to future work.

Internal validity refers to causation: are changes in the dependent variable necessarily the result of manipulations to treatments. In our case the treatments are snippets that differ in use of constructs and in the structure of conditionals. However they may also differ in length, MCC, or some other metric, which may have an effect.

Another problem is possible learning effects. Since most of the snippets actually perform the same logic, there is a threat of a learning effect with the progress from one question to the next. This is mitigated by randomizing the order, and moreover observed subjects did not notice the commonality of the snippets.

The experiment can be conducted anywhere and at any time. Not all of the subjects necessarily took the experiment under the same conditions. They could do all the questions in a row or take a break for a long time in the middle. They could use accessories without us knowing. Even though we added the "skip" button, we can not actually know whether a subject just guessed when giving some answers.

Finally there is the personality of the subjects, and its effect on the way they choose to deal with the snippets. The time is limited, but still a variation in the way different subjects answered was observed. There were subjects that chose to check themselves, while other subjects answered immediately when they thought they were correct and moved on.

External validity refers to generalization. The snippets used are synthetic code, created just for the experiment. The generalization to real production code is therefore questionable, especially since our snippets deal only with

TABLE III
MAIN RESULTS FOR THE DIFFERENT RESEARCH QUESTIONS

RQ	*description*	*results*
1	`if` vs. `for`	`for` loops are harder than `if`s
2a	expression size	3 predicates is harder than 2 3 vs. 4 not significant
2b	compound vs. structure	differences not statistically significant
2c	flat vs. nesting	flat structures appear to be slightly easier
2d	negation	some but not all uses of negation are harder: negations are different from each other
3	loop idioms	loops counting up are easier unusual loop bounds do not have a significant effect

finding a number in a set of ranges, and do not necessarily pertain to the general issues of constructs, conditionals, etc. The justification is that we preferred to limit the experiment to a narrow scope in order to establish a solid base that allows for future expansion.

Another concern is that the experimental subjects may not be a valid sample. They all come from the same companies, and even certain departments in them. Thus replications with other subjects and additional code samples are as always needed.

IX. Conclusions and Future Work

How to measure code complexity — and even how to define code complexity — is a contentious issue. Many different metrics have been suggested, each focusing on certain specific aspects of the code. But there has been relatively little empirical evidence to support such metrics and to compare them to each other.

We have designed and implemented an experimental platform, fashioned as an online game, which can be used to measure the speed and accuracy of interpreting code snippets. We used this to measure the performance of 222 professional programmers as they interpret up to 14 different code snippets from a pool of 40 such snippets, that have diverse structures.

Analyzing the results we find that indeed different code structures take different times to interpret. For example, our results indicate that `for` loops take more time than sequences of `if`s. Thus the approach taken by MCC, for example, where all branching constructs are given the same weight, is overly simplistic. Moreover, we also found differences that stem from different ways to express the same logical conditions (e.g. different ways of using negation), or from adhering to or violating common idioms (e.g. that loops count up). This implies that looking only at basic syntactic constructs is too limited. The main results are summarized in Table III.

While these results are illuminating and demonstrate new paths for empirical investigation, they are far from being comprehensive. Our study focused on one specific family of conditions, and a limited number of structures that can be used to express them. We did not cover `while` loops, `switch case`s, conditionals with equality and inequality, and much more. A lot of additional work will be needed to complete the picture and better quantify the effects of different structures and the interactions between them.

Once such additional work is conducted, it may be possible to derive sound complexity metrics that are better than those available today and are backed by empirical data. For example, instead of just counting constructs it may be possible to weight them, and perhaps also modify the weights based on nesting and other context [23].

To start with, we are already planning additional experiments that focus on different styles of negation and use of De Morgan's laws, and on the effect of different levels of nesting. We are also planning to reproduce this work using another domain, such as array and string operations, to improve external validity. On the methodological front, we note that anecdotal evidence from our subjects suggests that they appreciated the gamification element of the experiment. To support this we have started another experiment aimed at assessing how much (if at all) the gamification elements indeed contribute to motivation and achievements, by re-running experiments with these elements removed.

On a wider scale, we note that the code snippets we use and the methodology in general do not distinguish between different levels of understanding, and specifically interpretation and comprehension. We are therefore planning new experiments that will establish this distinction. This will be a first step in addressing the deeper issues of what affects understanding and how to aid comprehension.

Verifiability

All experimental materials, including the source code for the gamified experimental platform and all versions of all code snippets, are available on github:
https://github.com/shulamyt/break-the-code/tree/icpc17.

Acknowledgments

Many thanks to Micha Mandel for his help with the statistical analysis. This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).

References

[1] A. Agresti and M. Kateri, *Categorical Data Analysis*. Springer, 2011.

[2] M. Ali and M. O. Elish, "*A comparative literature survey of design patterns impact on software quality*". In *Intl. Conf. Inf. Sci. & App.*, Jun 2013, DOI: 10.1109/ICISA.2013.6579460.

[3] V. Arunachalam and W. Sasso, "*Cognitive processes in program comprehension: An empirical analysis in the context of software reengineering*". *J. Syst. & Softw.* **34(3)**, pp. 177–189, Sep 1996, DOI: 10.1016/0164-1212(95)00074-7.

[4] T. Ball and J. R. Larus, "*Using paths to measure, explain, and enhance program behavior*". *Computer* **33(7)**, pp. 57–65, Jul 2000, DOI: 10.1109/2.869371.

[5] G. R. Bergersen, D. I. K. Sjøberg, and T. Dybå, "*Construction and validation of an instrument for measuring programming skill*". *IEEE Trans. Softw. Eng.* **40(12)**, pp. 1163–1184, Dec 2014, DOI: 10.1109/TSE.2014.2348997.

[6] R. Brooks, "*Towards a theory of the comprehension of computer programs*". *Intl. J. Man-Machine Studies* **18(6)**, pp. 543–554, Jun 1983, DOI: 10.1016/S0020-7373(83)80031-5.

[7] R. P. L. Buse and W. R. Weimer, "*A metric for software readability*". In *Intl. Symp. Softw. Testing & Analysis*, pp. 121–130, Jul 2008, DOI: 10.1145/1390630.1390647.

[8] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "*Exploring the influence of identifier names on code quality: An empirical study*". In 14th *European Conf. Softw. Maintenance & Reengineering*, pp. 156–165, Mar 2010, DOI: 10.1109/CSMR.2010.27.

[9] B. Curtis, "*Substantiating programmer variability*". *Proc. IEEE* **69(7)**, p. 846, Jul 1981, DOI: 10.1109/PROC.1981.12088.

[10] B. Curtis, J. Sappidi, and J. Subramanyam, "*An evaluation of the internal quality of business applications: Does size matter?*" In 33rd *Intl. Conf. Softw. Eng.*, pp. 711–715, May 2011, DOI: 10.1145/1985793.1985893.

[11] B. Curtis, S. B. Sheppard, and P. Milliman, "*Third time charm: Stronger prediction of programmer performance by software complexity metrics*". In 4th *Intl. Conf. Softw. Eng.*, pp. 356–360, Sep 1979.

[12] G. Denaro and M. Pezzè, "*An empirical evaluation of fault-proneness models*". In 24th *Intl. Conf. Softw. Eng.*, pp. 241–251, May 2002, DOI: 10.1145/581339.581371.

[13] S. Deterding, D. Dixon, R. Khaled, and L. Nacke, "*From game design elements to gamefulness: Defining "gamification"*". In *15th Intl. Academic MindTrek Conf.: Envisioning Future Media Environments*, pp. 9–15, 2011, DOI: 10.1145/2181037.2181040.

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

[15] 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.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elementsof Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[17] G. K. Gill and C. F. Kemerer, "*Cyclomatic complexity density and software maintenance productivity*". *IEEE Trans. Softw. Eng.* **17(12)**, pp. 1284–1288, Dec 1991, DOI: 10.1109/32.106988.

[18] V. Gruhn and R. Laue, "*On experiments for measuring cognitive weights for software control structures*". In 6th *Intl. Conf. Cognitive Informatics*, pp. 116–119, Aug 2007, DOI: 10.1109/COGINF.2007.4341880.

[19] 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.

[20] 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.

[21] K. Huotari and J. Hamari, "*Defining gamification: A service marketing perspective*". In *16th Intl. Academic MindTrek Conf.*, pp. 17–22, 2012, DOI: 10.1145/2393132.2393137.

[22] E. R. Iselin, "*Conditional statements, looping constructs, and program comprehension: An experimental study*". *Intl. J. Man-Machine Studies* **28(1)**, pp. 45–66, Jan 1988, DOI: 10.1016/S0020-7373(88)80052-X.

[23] A. Jbara and D. G. Feitelson, "*How programmer read regular code: A controlled experiment using eye tracking*". *Empirical Softw. Eng.* DOI: 10.1007/s10664-016-9477-x. (to appear).

[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] B. Katzmarski and R. Koschke, "*Program complexity metrics and programmer opinions*". In 20th *Intl. Conf. Program Comprehension*, pp. 17–26, Jun 2012, DOI: 10.1109/ICPC.2012.6240486.

[26] M. Klerer, "*Experimental study of a two-dimensional language vs Fortran for first-course programmers*". *Intl. J. Man-Machine Studies* **20(5)**, pp. 445–467, May 1984, DOI: 10.1016/S0020-7373(84)80021-8.

[27] S. Letovsky, "*Cognitive processes in program comprehension*". *J. Syst. & Softw.* **7(4)**, pp. 325–339, Dec 1987, DOI: 10.1016/0164-1212(87)90032-X.

[28] P. Mair and R. Hatzinger, "*Extended Rasch modeling: The eRm package for the application of IRT models in R*". *J. Stat. Softw.* **20(9)**, May 2007, DOI: 10.18637/jss.v020.i09.

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

[30] J. C. Munson and T. M. Khoshgoftaar, "*Applications of a relative complexity metric for software project management*". *J. Syst. & Softw.* **12(3)**, pp. 283–291, Jul 1990, DOI: 10.1016/0164-1212(90)90051-M.

[31] G. J. Myers, "*An extension to the cyclomatic measure of program complexity*". *SIGPLAN Notices* **12(10)**, pp. 61–64, Oct 1977, DOI: 10.1145/954627.954633.

[32] B. T. Mynatt, "*The effect of semantic complexity on the comprehension of program modules*". *Intl. J. Man-Machine Studies* **21(2)**, pp. 91–103, Aug 1984, DOI: 10.1016/S0020-7373(84)80060-7.

[33] N. Ohlsson and H. Alberg, "*Predicting fault-prone software modules in telephone switches*". *IEEE Trans. Softw. Eng.* **22(12)**, pp. 886–894, Dec 1996, DOI: 10.1109/32.553637.

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

[35] J. Rilling and T. Klemola, "*Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics*". In 11th *IEEE Intl. Workshop Program Comprehension*, pp. 115–124, May 2003.

[36] H. Sackman, W. J. Erikson, and E. E. Grant, "*Exploratory experimental studies comparing online and offline programming performance*". *Comm. ACM* **11(1)**, pp. 3–11, Jan 1968, DOI: 10.1145/362851.362858.

[37] 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.

[38] 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.

[39] M. Shepperd, "*A critique of cyclomatic complexity as a software metric*". *Software Engineering J.* **3(2)**, pp. 30–36, Mar 1988, DOI: 10.1049/sej.1988.0003.

[40] B. Shneiderman and R. Mayer, "*Syntactic/semantic interactions in programmer behavior: A model and experimental results*". *Intl. J. Comput. & Inf. Syst.* **8(3)**, pp. 219–238, Jun 1979, DOI: 10.1007/BF00977789.

[41] E. Soloway and K. Ehrlich, "*Empirical studies of programming knowledge*". *IEEE Trans. Softw. Eng.* **SE-10(5)**, pp. 595–609, Sep 1984, DOI: 10.1109/TSE.1984.5010283.

[42] 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.

[43] A. von Mayrhauser and A. M. Vans, "*Program comprehension during software maintenance and evolution*". *Computer* **28(8)**, pp. 44–55, Aug 1995, DOI: 10.1109/2.402076.

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