

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

Shulamyt Ajami · Yonatan Woodbridge ·  
Dror G. Feitelson

Received: date / Accepted: date

**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 220 professional programmers can interpret code snippets with similar functionality but different structures; snippets that take longer to understand or produce more errors are considered harder. 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. We also found that the metrics of time to understanding and errors made are not necessarily equivalent. Thus loops counting down took slightly longer, but loops with unusual bounds caused many more errors. By amassing many more empirical results like these it may be possible to derive better code complexity metrics than we have today, and also to better appreciate their limitations.

**Keywords** Code complexity · program understanding · gamification

---

Dror Feitelson holds the Berthold Badler chair in Computer Science.  
This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).  
This paper is an invited extended version of a paper from ICPC 2017.

S. Ajami and D. G. Feitelson  
Department of Computer Science  
The Hebrew University, 91904 Jerusalem, Israel  
E-mail: shulamyt@gmail.com, feit@cs.huji.ac.il  
Y. Woodbridge  
Department of Statistics  
The Hebrew University, 91905 Jerusalem, Israel  
E-mail: yonatan.woodbridge@mail.huji.ac.il

## 1 Introduction

Program comprehension is largely about bridging gaps of knowledge [15]. 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 (`gotos` 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 indented for readability?), and more [24, 35, 50, 22, 16, 17, 20, 36].

Once factors influencing code 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 [49]. 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 [68, 78, 29, 8, 39].

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` [59], and so on. However, proposals to give different constructs different weights did not report empirical evidence supporting the proposed weights [66, 31]. 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 functionality can be expressed in a wide variety of ways, using different syntactical constructs and conditionals, thereby enabling meaningful comparisons; at the same time, it is simple and straightforward and does not require any prior domain knowledge. 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 220 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. Measuring the time needed to interpret snippets and the number of mistakes made provides an operationalization of how hard each one is.
- 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).
- Indication that the metrics of time to understanding and error rate in understanding may measure different effects: making mistakes may reflect misconceptions rather than complexity.
- Demonstration of a small learning effect during the experiment as subjects become more proficient at answering such questions. At the same time a larger effect is due to experimental subjects experiencing difficulties dropping out.
- Analysis indicating that most demographic variables (such as experience and sex) do not affect the results.

This paper is an invited extended version of a paper with the same title that appeared in the 25<sup>th</sup> *International Conference on Program Comprehension* [4]. The main additions are in the deeper analysis of the experimental results: The analysis of how the time to a correct answer and the error rate depend on the serial number of the snippet in the experiment, demonstrating a small learning effect (7.1); The analysis of the correlation (or lack thereof) between the time to a correct answer and the error rate for different snippets (Section 7.4); And the analysis of the possible effect of the demographic characteristics of the experimental subjects (Section 8). In addition, many details have been added throughout, and the structure of the paper has been improved to make it easier to read. As part of this all the code snippets used are displayed as part of the discussion on selecting them.

## 2 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 [49]. He also suggested an extended version where the components of compound conditionals are enumerated separately, which in a sense anticipates one aspect

of 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” [51]. Over the years there have been some reports that this and related metrics can be used to predict code quality [21, 50, 56, 65], but also reports that claimed that any correlations with these metrics are low or nonexistent [22, 25, 42]. In particular, there has been a debate on the degree to which cyclomatic complexity adds upon the straightforward code length metric [36, 45, 28]. And as noted above, proposals to give different constructs different weights did not report empirical evidence supporting the proposed weights [66, 31].

Other studies did compare specific constructs empirically. Mynatt considered how the use of iteration vs. recursion affects comprehension, as measured by ability to recall programs [53]. 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 [38]. We focus more on how (compound) conditionals are expressed, and on quantifying this effect.

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 the structure of a certain code fragment, this understanding can be leveraged for its repetitions [75, 39, 55]. At this stage our experiments consider only basic structures in isolation.

We define programming idioms as commonly used coding constructs, such as the double loop used to traverse a 2-D image. There has been little study of the effect of using (or violating) such basic idioms on program comprehension, and most related work appears to be at a slightly higher level of abstraction. One of the examples is the work of Soloway and Ehrlich in the early 1980s [72], in the context of studying the differences between experts and novices. They define “programming plans” as “generic program fragments that represent stereotypic action sequences in programming”, allowing programmers to perform “chunking” and to identify standard approaches to problem solving [62]. One of the findings was that experts know of and exploit “rules of discourse” and “accepted conventions of programming”. Hansen et al. also show that programs that conform to expectations, as opposed to programs that contain misleading elements, are easier to understand correctly [33].

Other studies have also aimed at theories of cognitive processes [69, 15, 46, 76, 6, 63]. For a brief exposition on those specifically related to programming plans see e.g. Parnin et al. and references therein [57]. We make no claim regarding the internal representations of programs or the cognitive processes behind them, and in our opinion much more experimental data is needed for such theorizing. As Hansen et al. also claim, the relationship between code, correctness, and speed is a complex one, and even “small notational changes can have profound effects” [33].

On the practical level repeated use of code fragments may be seen as a precursor of design patterns, which were later popularized by the “gang of four” in the context of object-oriented programming [27]. 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 [5]. 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 for the interaction of code comprehension with demographic variables, there has been significant work comparing code comprehension by experts and novices, oftentimes in the context of evaluating a certain methodology or tool [41, 2, 9, 74, 1]. Interestingly, the effect seems to be non-linear: differences were found mainly between novices and those with some training, and less so between professionals with different degrees of experience [64, 73, 10]. These results concur with the “laws of practice”, which state that initially, when one lacks experience, every additional bit of experience has an important effect, but when one is already experienced, the marginal benefit from additional experience is much reduced [54, 34]. Other work has shown that sex may lead to differences in the approach to understanding code [67].

On the methodological front, an important issue is how to operationalize the measurement of code complexity. We follow many previous studies in requiring experimental subjects to answer questions about code snippets. In particular, we require subjects to deduce what the code will print as its output. To further the experimental subjects’ engagement with the experiment we use gamification. This is motivated in part by an analogy to notions like considering programming as a sporting event [13, 43].

Previous work on using gamification to enhance motivation and engagement has focused mainly on education, and especially online learning platforms [32]. To the best of our knowledge, the only prior use of gamification in the context of empirical experiments is that by Yoder and Belmonte [79].

### 3 Research Questions

The study described actually has two main parts. The *goal* of the first and major part is to measure how different syntactic and other factors influence code complexity and comprehension. The second part concerns the interaction of demographic explanatory variables with the code-related explanatory variables. To concretize the main goal, we focused on the following specific *research questions*. Each of these questions identifies a distinct potential code feature and asks about its effect on code complexity and comprehension:

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 output printed by these snippets. A longer time or more errors are assumed to reflect difficulties in understanding the code, and hence serve to identify more complex (harder) code.

It should be noted, however, that it is not necessarily self-evident that time and correctness measure the same aspect of code complexity. Looking at them separately may therefore lead to interesting observations. This leads to another research question:

4. What if any is the correlation between the metrics of time to correct answer and error rate?

As noted our main goal is to characterize the elements of code complexity. But code complexity may be in the eye of the beholder. Therefore a *secondary goal* was to study the interaction of code complexity and demographic variables. In particular, we wanted to check the following *research questions*:

- 5. Do experimental subjects experience a learning effect during the experiment?
- 6. What if any is the effect of demographic variables on the performance of experimental subjects faced with code comprehension tasks, and specifically
  - (a) What is the effect of age?
  - (b) What is the effect of sex?
  - (c) What is the effect of education (having an academic degree)?
  - (d) What is the effect of experience?
  - (e) Does performance correlate with self-assessment?

The *metrics* used were the same as those used above, namely the time and correctness of answering about the different code snippets. But in analyzing the collected data, we compared the performance of different groups of experimental subjects rather than the performance on different groups of snippets. Likewise, to investigate the learning effect, we compare performance on questions based on their serial number rather than based on their content.

## 4 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.

## 4.1 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, with natural scaling to different sizes, and using or violating idioms.

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 does not 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.

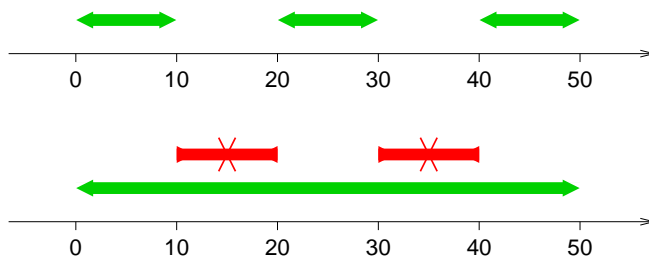
Third, will the code snippets be synthetic or real? Obviously it is better to base an experiment on snippets taken from a real program in the interest of external validity. But it may be hard to find suitable code snippets that have common ground as we desire, leading to noise that might cloak the potentially subtle effect we want to measure.

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. We make a distinction between *interpretation* and *comprehension*. Interpretation — as in an interpreter which executes code one instruction at a time — is being able to trace the execution of the code and find its output. It requires an understanding of the instructions and the memory model, but does not necessarily reflect a high-level understanding of what the program does. Comprehension, in contradistinction, means being able to state the objective of the code. Such understanding can be achieved in various ways. One is a top-down process in which hypotheses about the program are refined as more details are studied. Alternatively a bottom-up process can be used, in which details are grouped together to achieve a more general understanding.

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 (or ever physically) execute the code to find its outcome. Thus it might be claimed that resorting to interpretation is a sign of complexity, or at least a lack of recognizing what is going on at a glance [62]. However,



**Fig. 1** example of different ways to express three ranges.

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. Our experiment requires finding the output printed by code snippets, and subjects are free to use whatever technique they wish to do so, whether mechanical interpretation or high-level comprehension. Investigating the distinction between interpretation and comprehension further is left to future work.

#### 4.2 Choosing a Common Framework

Taken together, the considerations listed above led to the decision to use a set of synthetic code snippets written especially for the experiments, 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, as this is a generic and simple operation that can serve as a prototype of more complicated decisions. Each range is defined by a conjunction of two simple Boolean atoms with “greater than” ( $>$ ) and “smaller than” ( $<$ ) comparisons, e.g.  $x > 0 \ \&\& \ x < 10$ . 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, and they are easy to scale (add more ranges). But most importantly, this functionality can be expressed in many different ways. For example, consider the question of verifying whether  $x$  is in any of three successive ranges  $(0,10)$ ,  $(20,30)$ , and  $(40,50)$ . As shown in the top of Figure 1 this can be expressed as a disjunction of predicates defining the three ranges. Alternatively, we can store the pairs of endpoints defining the ranges in an array, and use a loop that checks these pairs one at a time. A different approach, shown in the bottom of Figure 1, is to use one predicate to select the whole range from 0 to 50, and conjugate this with negations of the gaps from 10 to 20 and from 30 to 40.



The code snippets used to express the different approaches include print statements that identify the path in the code. For example, the simple disjunction will be

```

if (x>0 && x<10 || x>20 && x<30 || x>40 && x<50) {
    print(1); /*in*/
} else {
    print(2); /*out*/
}

```

Note that 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.

### 4.3 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.

#### 4.3.1 Version a: Simple Disjunction

We start with the simplest and most straightforward version, which we call version a. This version uses the simple disjunction introduced above. The first variant of this version does this in a single compound logic expression, which for 3 ranges is as follows:

```

a1: (x>0 && x<10 || x>20 && x<30 || x>40 && x<50)

```

This snippet is referred to in the sequel as a1, which stands for version “a” variant “logic”.

The second variant achieves the same effect using a structure of nested if expressions. In order to be able to compare snippets consistently, we define precise conversion rules for breaking compound predicates. Conjunction (&&) in a compound expression converts to an additional if nested in the then block:

```

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

```

Disjunction (||) converts to an if nested in the else block:

```

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

```

Note, however, that this cannot be completely automatic, as the ordering of the atoms needs to be decided so as to preserve correctness. Brackets are added only if necessary, meaning that the expression outcome will be different without brackets.

Using these conversion rules, snippet `al` can be reshaped using nested ifs with simple predicates. This leads to snippet `as` (version “a”, variant “structure”):

```
as:  if (x>40) {
      if (x<50) {
        print ("1");
      } else {
        print ("2");
      }
    } else if (x>20) {
      if (x<30) {
        print ("3");
      } else {
        print ("4");
      }
    } else if (x>0) {
      if (x<10) {
        print ("5");
      } else {
        print ("6");
      }
    } else {
      print ("7");
    }
}
```

Note that while these two variants are equivalent in terms of checking whether the parameter `x` is in any of the ranges (0, 10), (20, 30), or (40, 50), they differ in many respects: snippet `as` is much longer and provides more detailed information on the code path followed, while snippet `al` is short but employs a much more complex logical expression. These issues are obvious threats to validity and are discussed in Section 9. For now it suffices to say that they are unavoidable compromises needed to express the selected functionality in diverse ways.

Note that both snippet `al` and snippet `as` can be scaled with different numbers of ranges. In our experiments we use 2, 3, or 4 ranges, for RQ 2a. The same applies for all the snippets described in the following subsections.

The next step is to create snippets based on different versions of code structure, with differences in nesting. These reflect RQ 2c. Just like version `a`, these additional versions can also be expressed in two main ways: using a single `if/else` statement with a compound logical expression (like snippet `al`), or using control flow with multiple `if/elses` with a single condition in each one (like snippet `as`). This reflects RQ 2b.

### 4.3.2 Version b: Two-Level

The first alternative to version a is 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. This is called version b. The logic-expression variant is as follows, using 4 ranges to emphasize the structure:

bl:

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

This logical expression can be broken into a structure of nested `ifs` using the conversion rules presented above. This leads to a structural variant of this snippet, called `bs`.

Note that the logic expression treats the range bounds in order, where each one is within the context of the previous one. This leads to very deep nesting. We therefore also defined an alternative two-level scheme in which the second level is flat. This variant is called `bl1`.

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

### 4.3.3 Version c: Recursive Structure

The second alternate version has a recursive structure. It first divides the whole range into two, and tests whether the number is in the first part or the second part. It then continues recursively inside the selected part, drilling down to the individual ranges. This leads to a more balanced nested structure, which will be called version c. The logic-expression variant is as follows, again using 4 ranges:

```
c1: (x>40 && (x<50 || x>60 && x<70) || x<30 && (x>20 || x<10 && x>0))
```

Just like the previous two versions, this logical expression too can be broken into a structure of nested `ifs` using the conversion rules presented above. This leads to a structural variant of this snippet, called `cs`.

### 4.3.4 Using Negation

The next step was to express the first variant (a1 above) with negation, in three different variants that will be denoted `an`, `an1`, and `an2`. This reflects RQ 2d. The logical expressions used, for 3 ranges, are as follows:

```
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))
```

**Table 1** Variations between code snippets that compare `for` loops

<i>version</i>	<i>init</i>	<i>cmp</i>	<i>end</i>	<i>step</i>
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	--

The first (snippet `an`) implements the construct shown at the bottom of Figure 1: it includes the whole range from the lowest bound to the highest bound, and negates the gaps. The second (snippet `an1`) is a flat version, and negates a disjunction of both the gaps and the rays below the bottom bound and above the top bound. The third (snippet `an2`) is obtained by applying De Morgan’s law to `an1`, negating each part individually and turning the disjunctions into conjunctions.

#### 4.3.5 Using Loops

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 `ifs` 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.

In fact we have two versions of snippets using `for` loops. The first is based on setting the ends of the ranges to be multiples of 10, say 0 to 10, 20 to 30, and so on. In this case we can express them by simple arithmetic manipulations of the `for` loop iteration variable. This variant is called `f*`:

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

Alternatively, it is possible to store the ranges’ end points in an array, and loop over this array to handle the ranges one after the other. This variant will be called `f[]`.

```
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]) { ... }
```

Note that these are both reasonable uses of `for` loops.

#### 4.3.6 Loop Idioms and Their Violation

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:

**Table 2** Code snippets used for each research question

<i>RQ</i>	<i>description</i>	<i>snippet comparisons</i>
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

- The idiomatic for loop for (i=0; i<n; i++).
- Loops with different end conditions, including 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--).

The precise definitions of the seven snippets we use are given in Table 1.

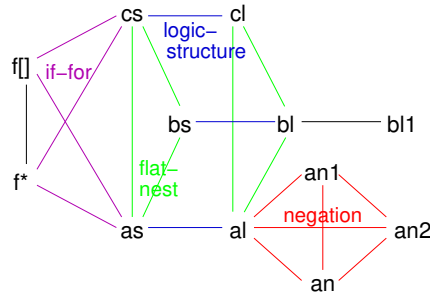
#### 4.4 Generating a Test Plan

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. A summary of these snippets, the notation used to identify them, and their relationship to research questions is given in Table 2. In the pilot study 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.

One possible course of action is to just select around 12 snippets at random for each subject. But (as explained below) this confounds differences due to experimental treatments with individual differences between the subjects, and reduces experimental power. The alternative is to assign pairs of snippets that we want to compare to the same subjects. For example, snippet al (simple logic expression) can be compared with as (an equivalent structure of ifs) 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. If we give such sets of snippets to the same subjects, we can use within-subject comparisons and hopefully reduce variance. The full graph showing all pairs of comparable snippets (except those relating to size) is given in Figure 2.

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, bl1, cs, cl}, and {al, an, an1, an2}. These facilitate the collection of data for within-subject comparisons. For each subject we pick one of these three groups at random.



**Fig. 2** Code snippet comparisons in relation to research questions.

- 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).
- Finally, we add three special idiom snippets drawn randomly.

The selected snippets are presented in a random order to avoid confounding with any learning effects. The total number of snippets presented to each subject is between 11 and 15.

## 5 Experimental Platform

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

### 5.1 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.

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 can we 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).

# Get the CODE?

This site is part of a research in software engineering which aims to measure the comprehension of different code snippets. Note that the goal is to characterize the different code snippets, not to evaluate the programmers!

During the game you will be given code snippets of varying difficulty drawn randomly from a pool. There is a time limit for each question. Try to answer every question as quickly as possible (and of course correctly, but please think it out and do not guess!). Remember that your score depends on how fast and how correctly you answered. The total time is expected to be about 15 minutes.

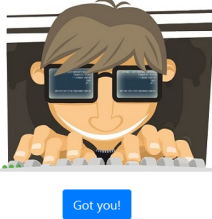
It is important to note:

We do not collect any identifying information (not even IP addresses). All information gathered is anonymous for research purposes only.

You may retire at any point (even though we'd be happy if you stay with us until the end).

Continuing to the questionnaire indicates agreement to participate in the research. The research is supervised by Prof. Dror Fetteison, Dept. Computer Science, Hebrew University. For any concerns contact: Shulamyt Ajami (shulamyt@gmail.com)

Good luck!



**Fig. 3** Introduction page of the experiment.

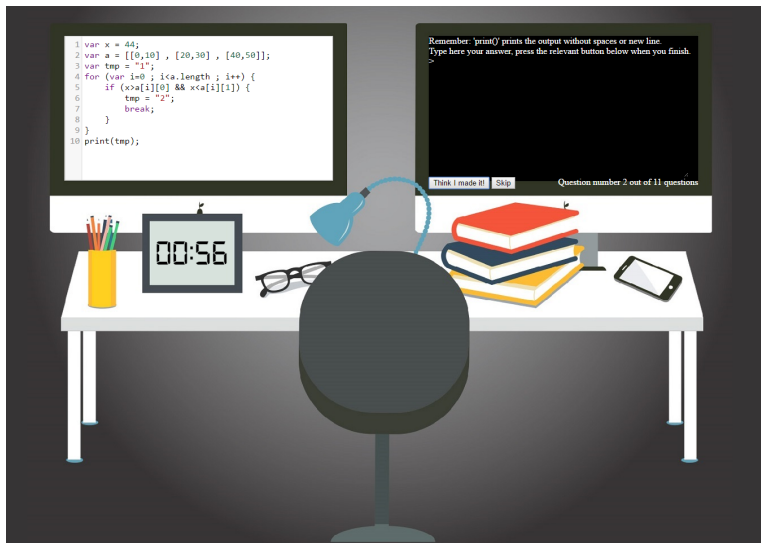
- **Methodological:** To motivate the subjects we design the website based on some *gamification* principles. Huotari and Hamari define gamification by the user experience, which should be fun and challenging [37]. Deterding et al. say that gamification is reflected in the system implementation, by including game elements like graphics, an avatar, a timer, a progress bar, feedback, etc. [23]. We will show how we implement gamification according to both these definitions.

Gamification is generally defined as a form of service packaging where a core service is enhanced by a rules-based service system that provides feedback and interaction mechanisms to the user with an aim to facilitate and support the user's overall value creation. One possible interpretation is to emphasize certain elements the system should include. But an alternative interpretation is to highlight the user experience. We favor the latter, as the user experience can be expected to more directly affect motivation, and note that our users indeed experienced the experiments as a game. This is in line with the notion that motivation is largely intrinsic, and that interest and a sense of fun are better motivators than monetary reward [58].

## 5.2 The Platform

We present the system by describing its flow. More details and screenshots are provided with the experimental materials of this paper.

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



**Fig. 4** Screenshot of gamified experimental platform.

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 3) 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. The next screen is a demographic questionnaire with details on education and experience. None of the fields are mandatory. Notice that choosing a test plan does not depend 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 4. 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:
  - 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 motivates 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 pace. In particular, subjects that take a part in the experiment 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.

The infrastructure used to run the experiment is as follows. We use DigitalOcean for cloud server. We set up a server with Ubuntu 16.04.3 x64 as an operating system, and install (or verify that we have) the following packages:

- Node.js - a JavaScript runtime
- npm - package manager for JavaScript
- pm2 - production process manager for Node.js
- git - distributed version control system
- PostgreSQL - database

The backend is written in JavaScript using Express which is a Node.js web application framework. In the client we used mainly a library named React which is a JavaScript library for building user interfaces. All the source code is available in GitHub so you can easily repeat the experiment or reuse this tool for other experiments. A detailed “how-to” with all the steps needed to set up the environment and run an experiment is included in the experimental materials.

## 6 Experiment Execution

The experiment collected results of 220 subjects using the gamified platform. These results were used to answer research questions 1 through 6, concerning code complexity and the variables affecting it.

### 6.1 Subjects

Experimental subjects were recruited using a combination of “convenience sampling” and “snowball sampling”. These are non-probabilistic sampling techniques, where experimenters recruit subjects from among their acquaintances, and the initial subjects help to recruit additional subjects. Specifically, we started 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. Note that all the subjects were professional developers from the hi-tech industry. 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 more than a week.

The main 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.) Two of the subjects did not submit any answers, so the results are actually from only 220 participants. Ten of these retired after getting a single question wrong, but 158 continued till the end. The average number of snippets done was 10.24.

In terms of demographics, 118 of the subjects were male and 102 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.

### 6.2 Variables

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

### 6.3 Statistical Methodology

#### 6.3.1 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 [64, 19, 44, 60]. This causes comparisons of results obtained from different experimental subjects to suffer from excessive variance. It confounds differences due to experimental treatments with individual differences between the subjects, and reduces experimental power.

A possible solution to this problem is to use within-subject comparisons and paired samples. Thus, when we want to compare performance on snippet  $s$  with performance on snippet  $t$ , we measure how the *same subjects* perform on both. We then analyze the within-subject differences rather than the raw results, as described next. This is supposed to factor out individual differences to a large degree. Our assignment of snippets to subjects as described above facilitates this design.

#### 6.3.2 Statistical Approach for Comparing Times for Answering Snippets

We wish to compare the times to reach a correct answer on pairs of related snippets, in the context of the research questions. 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. the snippets in one use negation and those in the other don't). We denote them by  $\mathcal{A} = (a_1, \dots, a_n)$  and  $\mathcal{B} = (b_1, \dots, b_n)$ . The terms  $a_k$  and  $b_k$  are snippets' identifiers. The snippets are ordered so that  $a_k$  and  $b_k$  are considered a pair (Figure 2). Note that the members of each vector are not necessarily all different, as we may want to compare the same snippet against several others. 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  $k = 1, \dots, n$ . 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,

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

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  $2 \times 10^4$  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  $k = 1, \dots, n$  (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  $T_k$ ,  $k = 1, \dots, n$ . Calculation of significance was done as previously, with  $2 \times 10^4$  random permutations.

### 6.3.3 Statistical Approach for Comparing Error Rates on Snippets

The above analyses ignore incorrect results. To analyze the error rates we use the Rasch model [3, 11]. Denote the ability of subject  $i$  by  $\theta_i$ , and the difficulty of snippet  $j$  by  $\beta_j$ . 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  $\beta_j$  increases, the probability of the event  $\{X_{i,j} = 1\}$  decreases. Note that the model is not identifiable, since adding a constant  $c$  to  $\theta_i$  and  $\beta_j$  does not change expression (2). We therefore add a restriction that the sum of all estimated  $\beta$ s be 0. To estimate the  $\beta$ s we use the conditional maximum likelihood (CML) approach, which conditions on the number of correct answers of each subject as a sufficient statistic for  $\theta_i$ .

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  $\Sigma$  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 [48].

### 6.3.4 Statistical Approach for Demographic Variables

Our goal here is to test whether a difference exists in the mean time of correct solution between two partitions of the subjects, e.g. between sexes. For that purpose, we consider each code snippet separately. Thus for code snippet  $j$  we have two sets of results:  $\mathcal{Y}_{m,j}$  and  $\mathcal{Y}_{w,j}$ , representing the times required by all male (and, respectively, female) subjects who succeeded to correctly solve the  $j$ th code snippet. Note that the sets may be of different sizes.

Denote by  $\overline{Y}_{m,j}$  and  $\overline{Y}_{w,j}$  the mean solution time of snippet  $j$ , for men and women, respectively. We then test  $N$  hypotheses on difference between means, namely:

$$\begin{aligned}\mathcal{H}_0 &: \overline{Y}_{m,j} = \overline{Y}_{w,j} \\ \mathcal{H}_1 &: \overline{Y}_{m,j} \neq \overline{Y}_{w,j}\end{aligned}$$

Where  $N$  is the number of snippets. For this it is natural to consider the well known Welch's t-test of unequal variances [77]. In case the data is not normally distributed, we apply the test only if 20 or more observations are available, in which case we can assume normality of the mean [47]. The t-test statistic for the  $j$ th snippet is given in the following formula:

$$T_j = \frac{\overline{Y}_{m,j} - \overline{Y}_{w,j}}{\sqrt{\frac{\text{Var}(\mathcal{Y}_{m,j})}{|\mathcal{Y}_{m,j}|} + \frac{\text{Var}(\mathcal{Y}_{w,j})}{|\mathcal{Y}_{w,j}|}}}, \quad (4)$$

where  $\text{Var}(\mathcal{Y}_{\cdot,j})$  is the unbiased sample variance: denoting  $k = |\mathcal{Y}_{\cdot,j}|$ , we have

$$\text{Var}(\mathcal{Y}_{\cdot,j}) = \frac{1}{k-1} \sum_{i=1}^k (Y_{i,j} - \overline{Y}_{\cdot,j})^2 \text{ for } Y_{i,j} \in \mathcal{Y}_{\cdot,j}.$$

Using the above statistic, we again employ a permutation test. Specifically, for each code snippet  $j$  we consider the union set  $\mathcal{Y}_{m,j} \cup \mathcal{Y}_{w,j}$  of all observations from both sexes. In each trial, we randomly split the set into two subsets, in a manner that preserves the proportion  $\frac{|\mathcal{Y}_{m,j}|}{|\mathcal{Y}_{w,j}|}$ , and assign each subset to one "sex". We then compute (4) for the permuted data. Applying  $10^4$  such trials to each code snippet, we obtain the empirical probability density function (pdf) of the t-test. Note that under the null hypothesis, the means should not differ. We then obtain the p-value, by computing the area under the empirical pdf, where the domain values are larger than the non-permuted data Welch's statistic (4), in absolute value. We also compare our result with the p-value computed by approximating the Welch statistic distribution to the T distribution, with degrees of freedom formula given in [77].

### 6.3.5 Linear Model

The above analyses are intended to verify only whether a difference exists between the means of two sets of results. An alternative approach is to fit a linear model that estimates the effect (if any) of each factor. We consider the following mixed effect linear model [52], where a logarithmic transformation is applied on the dependent variable of time, as measured in milliseconds. A log model is preferred to improve the fit to required statistical assumptions, such as homoskedasticity (homogeneity of variance) and normality.

$$\log Y_{i,j} = \mu + u_i + \eta_j + \epsilon_{i,j}$$

$$u_i = \beta_1 \times \mathbb{1}_{fem_i} + \beta_2 \times \mathbb{1}_{age_i > 29} + \beta_3 \times \mathbb{1}_{exp_i > 4} + \beta_4 \times \mathbb{1}_{self_i > 3} + \beta_5 \times \mathbb{1}_{deg_i} + \zeta_i$$

In this model, the intercept  $\mu$  can be interpreted as the log of the average time it takes a male, under age 30, with four or less years of experience, of self-assessment lesser than 4, with no academic degree, to answer a snippet correctly.

The variable  $u_i$  relates to subject  $i$ , and depends linearly on demographic variables, as well as a random variable  $\zeta_i$ . We use indicator variables for the demographic factors, with the following coefficients:

- $\beta_1$  effect when the subject is female.
- $\beta_2$  effect when the subject is older than 29 years.
- $\beta_3$  effect when the subject has more than 4 years of professional experience.
- $\beta_4$  effect when the subject self assessment is greater than 3, in a scale of 1–5.
- $\beta_5$  effect when the subject has an academic degree<sup>1</sup>.

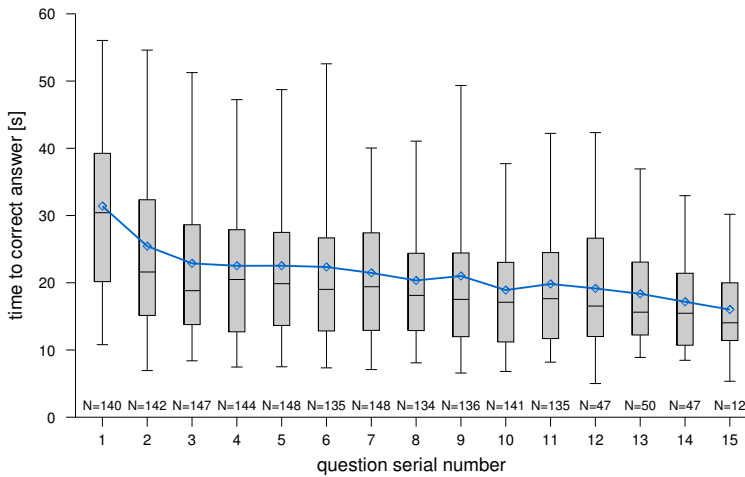
Since the dependent variable is the log transformation of time, each coefficient indicates the change of answer time in percents. The threshold values were selected based on observing the distributions of the respective variables. The criteria used were that the threshold should be near the median, and that if the histogram has a bimodal (or multi-modal) structure it should be between modes. Additional random variables  $\eta_j$  and  $\epsilon_{i,j}$  correspond to question  $j$  (snippet) and independent random noise of answer time, respectively.

## 7 Results for Complexity Effects

### 7.1 Learning Effects

Well-known threats to the validity of experiments involving a sequence of questions or tasks are learning effects and fatigue. Learning means that subjects become better at answering questions and performing tasks as they gain experience with the experiment. Fatigue means that they become tired with the experiment, leading to careless answers or guesses. Assessing whether such effects exist in our experiment was RQ 5.

<sup>1</sup> Subjects who did not answer the academic degree questions were assigned to the group of no degree.

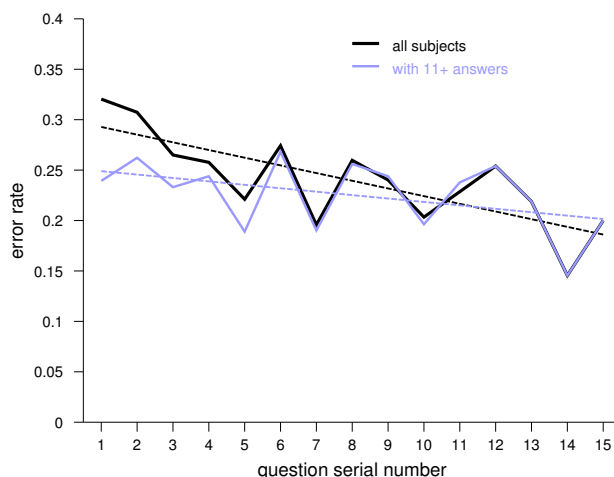


**Fig. 5** Distributions of time to correct answer for each question in the experiment. Each boxplot displays the 5th, 25th, 50th, 75th, and 95th percentiles. The line connects the average times.

To check for such effects we tabulated the time to correct answers as a function of the serial number of the question, regardless of what snippet the question was about. Thus we collected the times for all cases in which any snippet was the first to be asked about, for all cases where it was the second, and so forth. The results are shown in Figure 5. There is an unmistakable downward trend, indicating that some effect is at play. A linear regression analysis indicates that the time to correct answer is reduced by 0.75 seconds on average with each additional question. The correlation coefficient between question serial number and time to correct answer is  $\rho = -0.899$ , leading to a highly significant p-value of  $<0.00001$ . Note, however, that in reality the reduction in time is not the same for all questions. The biggest reduction in time to correct answer occurs in the first two questions. This probably reflects the time it takes experimental subjects to become acquainted with the experimental environment. In subsequent questions the difference is smaller, but the downward trend continues as subjects become more proficient at answering this type of questions. If we perform the regression analysis on only questions 3–15, the result is a reduction of 0.544 seconds with each additional question, and the correlation coefficient indicates a higher correlation, reaching  $\rho = -0.964$ .

However, there are actually three distinct effects that may affect the observed performance:

- Subjects learn how to answer the presented questions, as outlined above, leading to improved performance with time.
- Subjects experience fatigue but hang on, leading to reduced performance with time.



**Fig. 6** Error rate for each question in the experiment.

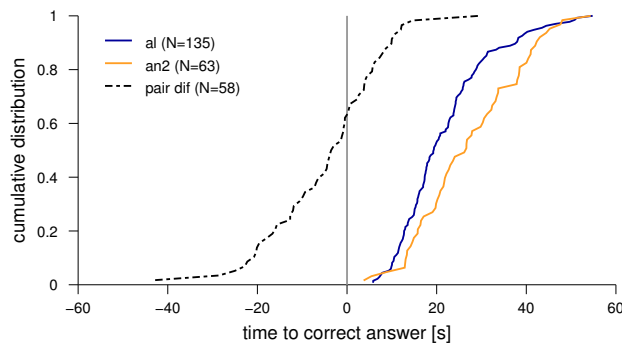
- Subjects experience difficulties or become bored, leading to dropping out before the end of the experiment.

To try and sort this out we tabulated the error rate as a function of the serial number of the questions: what fraction of the answers to the first question were wrong, what fraction of the answers to the second question were wrong, and so on. This was repeated twice, first for all subjects, and again using data only from subjects who gave answers to 11 questions or more (effectively identifying subjects who continued to the end).

The results are shown in Figure 6. When all subjects are considered an obvious downward trend is observed, and a linear regression indicates that the error rate is reduced by 0.0076 with each additional question. The correlation coefficient is  $\rho = -0.7651$ , and the p-value 0.001, so this is statistically significant. However, when subjects who dropped out before the end are excluded, this trend is much weaker. The error rate is reduced by 0.0034 on average with each additional question, and the correlation coefficient is only  $\rho = -0.442$ , leading to a p-value of 0.099 (not statistically significant at the 0.05 level).

This indicates that the reduced error rate was *not* due to experimental subjects who become better as the experiment unfolds. Rather, the correct interpretation seems to be that the apparent improved performance is mainly due to the fact that more and more subjects that tended to make mistakes dropped out. The subjects that continued with the experiment were those who made fewer mistakes, so the error rate was reduced. When combined with the above results concerning time to correct answer, it seems that there is no fatigue effect. But subjects probably do become slightly more proficient at answering quickly. And indeed, repeating the analysis of Figure 5 for only





**Fig. 7** Distributions of time to correct answer for two snippets.

those subjects that answered 11 questions or more, the results are practically the same (time reduced by 0.75 seconds per question and  $\rho = -0.882$ ).

The fact that there is a consistent trend of lower time to correct answer with question serial number raises the possibility of normalizing the results, thereby reducing their variance. This could potentially improve experimental power. We checked this but found that it did not lead to any improvements in the results. The following results therefore do not make any adjustment to the measured times.

## 7.2 Descriptive Statistics

An example of the raw results is shown in Figure 7. 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 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.

## 7.3 Results of Statistical Analysis

Results comparing various groups of snippets are given in Tables 3 and 4. Table 3 shows the results for performance in terms of time to correct answer. In addition to the mean and standard deviations it includes effect sizes, defined as the standardized mean difference: the difference between the mean times for

the two groups of snippets, divided by the standard deviation of these times. An effect size of 0.5 is considered “medium”, and an effect size of 0.8 is “large” [18]. It also includes p-values from permutation and Wilcoxon tests, using the statistical analysis described in Section 6.3.2.

Table 4 shows the results for error rates. It includes p-values from contrasts on the number of wrong answers, using the statistical analysis described in Section 6.3.3. When fitting the Rasch model, 10 subjects were eliminated because they had only one (wrong) answer. Note that the statistical significance results in the two tables 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 actually holds (type I errors). As we perform 31 tests, we may therefore expect 1 or 2 of them to pass. But the results were that no less than 16 (for performance) or 15 (for errors) were below the 0.05 threshold, rather than only 1 or 2. We may therefore conclude that the vast majority of these cases are indeed statistically significant.

An alternative approach is to apply the Bonferroni correction, by dividing the threshold of 0.05 by the number of tests performed. Then there is at most a 5% chance that some test will pass at random. The results were that 11 tests (for performance) and 7 tests (for errors) were below the Bonferroni corrected threshold.

### 7.3.1 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* (see Section 4.3.1) and *cs* (Section 4.3.3), and use a nested sequence of *>* or *<* tests to establish inclusion in a set of ranges. The latter do it with loops, using either arithmetic on the loop index in version *f\**, or an array that holds the range bounds in version *f[]* (Section 4.3.5).

The snippets using *for* took more than twice as long to interpret, and led to more errors. All the differences in performance were statistically significant (Table 3), and the effect sizes were also extremely large. Some of the differences in error rates were also statistically significant (Table 4). These results imply 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, while snippet *f[]* led to a more errors by a similar margin. Note, however, that a direct comparison was not statistically significant in both cases. Therefore the interpretation of these differences between variants requires additional study and more measurements.

**Table 3** Performance results of comparisons related to each research question, based on the methodology of Section 6.3.2

<i>RQ</i>	<i>compare this...</i>		<i>...with this</i>		<i>N</i>	<i>sd</i>	<i>effect</i>		<i>p-value</i>	
	<i>snippet</i>	<i>avg±stdv</i>	<i>snippet</i>	<i>avg±stdv</i>			<i>size</i>	<i>permut.</i>	<i>Wilcoxon</i>	
1	f*,f[]	40.5±17.8	as,cs	17.8±9.8	79	1	1.808	<10 <sup>-5</sup> **	<10 <sup>-5</sup> **	
if vs.	f*	46.6±19.0	as	18.3±11.2	24	1	2.300	<10 <sup>-5</sup> **	<10 <sup>-5</sup> **	
for	f[]	35.0±14.7	as	15.9±11.1	19	1	1.522	0.0001**	<10 <sup>-5</sup> **	
	f*	43.6±19.3	cs	20.3±9.3	19	1	2.076	0.0001**	0.0003**	
	f[]	34.5±15.0	cs	16.3±6.4	17	1	1.264	0.0001**	0.0001**	
	f*	45.6±21.9	f[]	33.2±13.2	14	2	0.775	0.0590	0.0782	
2a	3 seg	24.0±13.1	2 seg	20.6±13.4	235	1	1.264	<10 <sup>-5</sup> **	<10 <sup>-5</sup> **	
size	4 seg	24.4±12.9	3 seg	21.5±11.7	226	1	0.222	0.742	0.0773	
2b	al,bl,cl	21.2±8.6	as,bs,cs	19.5±9.6	115	1	0.281	0.0450*	0.0273*	
expr vs.	al	19.2±8.4	as	17.1±8.8	43	1	0.433	0.0924	0.0950	
struct	bl	22.5±9.4	bs	19.9±9.4	38	1	0.213	0.0647	0.0530	
	cl	22.3±7.7	cs	22.0±10.3	34	1	0.192	0.426	0.356	
	bl	23.7±11.0	bl1	21.6±9.0	35	1	0.151	0.129	0.171	
2c	as,al	17.5±7.9	bs,bl	21.1±10.2	84	2	-0.127	0.0036*	0.0032*	
flat vs.	as	16.6±8.1	bs	19.9±9.6	42	2	-0.301	0.0491*	0.0602	
nested	al	18.5±7.6	bl	22.4±10.7	42	2	-0.077	0.0249*	0.0206*	
	as,al	17.4±9.6	cs,cl	21.4±9.2	102	2	-0.104	<10 <sup>-5</sup> **	<10 <sup>-5</sup> **	
	as	16.7±9.9	cs	20.8±10.1	63	2	-0.321	0.0016*	<10 <sup>-5</sup> **	
	al	18.6±9.0	cl	22.4±7.4	39	2	-0.053	0.0191*	0.0140*	
	bs,bl	21.1±9.7	cs,cl	22.2±9.2	70	2	0.028	0.482	0.269	
	bs	20.1±9.7	cs	22.0±10.6	38	2	-0.014	0.343	0.383	
	bl	22.2±9.8	cl	22.3±7.5	32	2	0.028	0.958	0.493	
2d	an,an1,an2	25.0±10.9	al	23.5±10.4	166	1	0.298	0.0651	0.150	
negation	an	21.5±8.4	al	23.5±10.7	64	1	-0.017	0.920	0.930	
	an1	26.7±11.9	al	24.0±11.0	44	1	0.445	0.0898	0.118	
	an2	27.5±11.7	al	23.0±9.8	58	1	0.508	0.0044*	0.0077*	
	an1	28.1±11.9	an	21.5±8.5	42	2	0.494	0.0008**	0.0003**	
	an2	27.6±12.1	an	21.2±7.1	55	2	0.558	0.0015**	0.0007**	
	an1	26.0±11.9	an2	26.1±11.7	41	2	-0.057	0.972	0.929	
3	lp2,lp3,lp4	15.0±9.4	lp0,lp1	15.5±9.0	103	1	-0.172	0.833	0.818	
loops	lp5,lp6	20.6±7.9	lp0,lp1	16.2±8.7	73	1	0.430	0.0003**	0.0003**	

\* denotes statistical significance  $p < 0.05$ \*\* denotes statistical significance also after Bonferroni correction  $p < 0.0016$ .

avg±stdv: of time to correct answer.

N: number of paired samples of correct answers. sd: one-sided/two-sided.

### 7.3.2 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. (Note: in these analyses  $N > 220$  because each subject typically had 2 relevant comparisons.) 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, but this was not significant after Bonferroni correction.

**Table 4** Errors results of comparisons related to each research question, based on the methodology of Section 6.3.3

<i>RQ</i>	<i>compare this...</i>				<i>...with this</i>				<i>p-value contrast</i>	
	<i>snippet</i>	<i>N</i>	<i>err</i>	$\beta_j$	<i>snippet</i>	<i>N</i>	<i>err</i>	$\beta_j$		<i>sd</i>
1	f*,f[]	80	0.387	2.326	as,cs	197	0.213	0.065	1	0.0003**
if vs.	f*	40	0.325	0.914	as	97	0.175	-0.231	1	0.0072*
for	f[]	40	0.450	1.412	as	97	0.175	-0.231	1	0.0002**
	f*	40	0.325	0.914	cs	100	0.250	0.296	1	0.0813
	f[]	40	0.450	1.412	cs	100	0.250	0.296	1	0.0049*
	f*	40	0.325	0.914	f[]	40	0.450	1.412	2	0.291
2a	3 seg	573	0.206	-0.786	2 seg	351	0.145	-5.131	1	0.0081*
size	4 seg	430	0.333	2.985	3 seg	738	0.236	0.965	1	0.175
2b	al,bl,cl	261	0.142	-1.812	as,bs,cs	255	0.208	-0.190	1	0.970
expr vs.	al	147	0.082	-1.756	as	97	0.175	-0.231	1	0.999
struct	bl	55	0.182	-0.305	bs	58	0.190	-0.255	1	0.535
	cl	59	0.254	0.249	cs	100	0.250	0.296	1	0.544
	bl	55	0.182	-0.305	bl1	56	0.196	-0.318	1	0.491
2c	as,al	244	0.119	-1.987	bs,bl	113	0.186	-0.560	2	0.0580
flat vs.	as	97	0.175	-0.231	bs	58	0.190	-0.255	2	0.961
nested	al	147	0.082	-1.756	bl	55	0.182	-0.305	2	0.0097*
	as,al	244	0.119	-1.987	cs,cl	159	0.252	0.545	2	<10 <sup>-5**</sup>
	as	97	0.175	-0.231	cs	100	0.250	0.296	2	0.175
	al	147	0.082	-1.756	cl	59	0.254	0.249	2	<10 <sup>-5**</sup>
	bs,bl	113	0.186	-0.560	cs,cl	159	0.252	0.545	2	0.120
	bs	58	0.190	-0.255	cs	100	0.250	0.296	2	0.240
	bl	55	0.182	-0.305	cl	59	0.254	0.249	2	0.297
2d	an,an1,an2	233	0.232	-0.265	al	147	0.082	-1.756	1	0.0002**
negation	an	80	0.162	-0.922	al	147	0.082	-1.756	1	0.0593
	an1	76	0.355	0.509	al	147	0.082	-1.756	1	<10 <sup>-5**</sup>
	an2	77	0.182	-0.383	al	147	0.082	-1.756	1	0.0030*
	an1	76	0.355	0.509	an	80	0.162	-0.922	2	0.0020*
	an2	77	0.182	-0.383	an	80	0.162	-0.922	2	0.273
	an1	76	0.355	0.509	an2	77	0.182	-0.383	2	0.036*
3	lp2,lp3,lp4	218	0.385	0.866	lp0,lp1	173	0.225	-0.077	1	0.0003**
loops	lp5,lp6	132	0.341	1.104	lp0,lp1	173	0.225	-0.153	1	0.0215*

\* denotes statistical significance  $p < 0.05$

\*\* statistical significance also after Bonferroni correction  $p < 0.0016$ .

N: number of samples. err: wrong answers rate.  $\beta$ : estimated difficulty. sd: one/two sided.

### 7.3.3 RQ 2b: Single expression vs. structure

As noted above in Section 4.3.1, compound logical expressions composed of many atoms may be converted into a nested structure of simple *ifs*. But which of these two structures is easier to handle? The results were that the nested structure took slightly less time on average, but led to slightly more errors. However, practically all the comparisons were not statistically significant, and effect sizes were small.

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 *ifs* 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 `ifs` is not significantly harder, even though these code snippets are much longer when counting LOC and have deeper nesting. This contradicts common wisdom regarding LOC and nesting as indicators of complexity.

#### 7.3.4 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. These differences are exhibited by code snippets `al`, `bl`, and `cl`, described in Sections 4.3.1 through 4.3.3. Similarly, there were versions with nested `ifs` that each contain a single atom, based on the conversion rules described above in Section 4.3.1.

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 flat `a` versions were slightly easier than the other two, and this was statistically significant or nearly so especially when comparing with the `c` versions, which have a structure similar to a full binary tree. The `b` versions, which have a comb-like structure, were not statistically significantly different from the `a` versions after Bonferroni correction. The `bl1` version, which attempts to eliminate the deep skewed nesting of the `bl` version, nevertheless was very similar to it.

#### 7.3.5 RQ 2d: Use of negation

The logic expressions in the three snippets using negation were shown in Section 4.3.4; they were compared with the basic `al` version, shown in Section 4.3.1.

The results were somewhat surprising. Comparing the positive version `al` to the three negation versions, there was a significant difference in times only when comparing with `an2`, but even this was too weak to pass Bonferroni correction. Moreover, the average time for `an` was actually a bit *shorter* than for `al` (although not statistically significantly different). As a result comparisons of `an` with `an1` and `an2` led to strongly statistically significant differences and to the largest effect sizes. Regarding correctness, it was snippet `an1` that had the worse error rate, and significant difference from `al`. Thus not every negation causes difficulties to the same degree. Detailed investigation of this effect is left to future work.

#### 7.3.6 RQ 3: Common loop idioms

A special set of 7 snippets concerned the details of `for` loops on arrays, as described in Section 4.3.6. The specifics were listed in Table 1.



A cursory observation indicates that there is no strong correlation, and indeed the Pearson correlation coefficient is 0.35. This indicates that the two variables may reflect different effects. And indeed, several interesting patterns seem to support this conjecture.

First, we note that if we focus on strictly equivalent snippets, namely all those that check for inclusion in exactly 3 ranges (snippets `al`, `as`, `bl`, `bs`, `cl`, `cs`, `an`, `an1`, `an2`, `f*`, and `f[]`), the correlation jumps to 0.61. We focus on the size 3 snippets because they have many more samples in comparison with the size 2 and 4 snippets.

Within this group, the two most extreme snippets are `f*` and `f[]`. But they are extreme in different ways: `f*` took the most time, while `f[]` led to the most errors but one. A possible interpretation is that subjects were cognizant of the effort involved in understanding the arithmetic operations on the loop index, and were therefore more careful. The array version appeared easier, so they were less careful and made more mistakes.

Finally, The special loop idiom snippets (`sp0` through `sp6`) appear to form a separate cluster, with relatively low answer times but a wide range of error rates. This is the pattern that one may expect if many subjects based their answer on the expectation that the common loop idiom would be followed, and therefore failed to notice the deviation from the common idiom. In other words, the snippets that deviate from the expected idiom are misleading and cause misunderstandings, which are manifested in wrong answers — but they do not take more time. This is similar to the situation with misleading variable names that give rise to mistakes in understanding code [7].

## 8 Results for Demographic Effects

Before starting the experiment, subjects fill out a demographic questionnaire. This is important because differences in demographic attributes can confound the results [71]. Recording demographic variables enables one to check that the experiment was appropriately randomized. In addition, using this data we can re-analyze the results to see if any of the demographic variables collected influences the results, or whether the demographic variables perhaps interact with the results. This is the topic of RQ 6.

Recall that the demographic variables we collected were:

- Sex
- Age
- Experience (years worked)
- self-assessment of programming skills (suggested by Siegmund et al. as more reliable than years of experience [70])
- Education (degrees and years studied)

The effect of these demographic variables was analyzed using the methodology described in Section 6.3.4. As an example, the results of using Welch's t-test to compare men and women are shown in Table 5. Of the 40 code snippets

**Table 5** Results of comparison of sexes on 34 code snippets.

<i>snip</i>	<i>size</i>	<i>men</i>		<i>women</i>		<i>p-value</i>	
		<i>N</i>	<i>avg±sd</i>	<i>N</i>	<i>avg±sd</i>	<i>permut</i>	<i>t-test</i>
al	4	26	22.0±9.8	31	21.2±8.1	0.75	0.74
an	4	19	28.2±12.4	20	23.6±6.5	0.17	0.16
an1	4	22	30.8±12.9	15	24.8±14.3	0.20	0.20
as	4	20	22.5±12.0	15	21.1±10.2	0.71	0.70
bl1	4	13	25.8±14.1	11	28.6±11.2	0.60	0.59
bs	4	14	24.3±10.2	16	23.0±10.5	0.75	0.74
cs	4	26	19.1±10.7	15	22.3±11.4	0.40	0.38
f*	4	19	36.4±23.4	8	41.4±28.1	0.65	0.67
al	3	71	22.8±11.8	64	20.5±8.2	0.18	0.18
an	3	36	21.5±8.2	31	21.6±8.5	0.97	0.96
an1	3	24	28.3±11.7	25	24.8±12.2	0.32	0.32
an2	3	33	27.4±11.3	30	27.0±12.2	0.88	0.88
as	3	48	17.2±9.7	32	17.5±10.7	0.91	0.90
bl1	3	25	20.0±8.9	20	22.4±8.7	0.36	0.36
bl	3	26	22.4±11.5	19	22.7±9.7	0.95	0.95
bs	3	26	21.3±11.4	21	19.2±8.0	0.48	0.47
cl	3	25	19.8±7.6	19	25.6±9.4	0.03*	0.04*
cs	3	48	19.6±9.0	27	22.0±10.6	0.34	0.33
f*	3	18	48.8±19.7	9	44.7±20.3	0.63	0.63
f[]	3	13	34.4±13.6	9	33.4±15.1	0.87	0.87
al	2	41	14.6±9.0	35	15.2±8.4	0.76	0.75
an	2	22	21.7±10.4	20	18.7±7.9	0.29	0.29
an1	2	25	25.8±13.2	15	19.6±5.8	0.05	0.05
an2	2	16	23.2±13.2	17	19.7±10.6	0.42	0.41
bs	2	21	16.6±7.2	12	19.7±10.5	0.40	0.38
f*	2	19	36.1±18.6	6	29.8±9.9	0.32	0.30
f[]	2	22	32.9±18.9	11	36.7±21.8	0.63	0.62
lp0	-	39	15.1±11.0	37	18.1±10.8	0.23	0.23
lp1	-	39	17.2±10.1	19	16.9±8.7	0.92	0.92
lp2	-	32	16.4±7.2	23	12.3±10.7	0.13	0.13
lp3	-	28	13.8±8.8	21	15.4±9.5	0.56	0.55
lp4	-	12	17.2±11.2	18	16.5±11.6	0.87	0.86
lp5	-	30	19.8±8.1	18	22.6±8.3	0.27	0.26
lp6	-	21	21.1±8.2	18	20.5±7.5	0.81	0.80

overall, we consider only code snippets containing 20 or more observations. Thus, 34 code snippets remain for the analysis. As we clearly see, only one result appears to be statistically significant at level 0.05, but this is expected with so many tests. None are significant with Bonferroni correction.

In order to test the hypothesis that there is no difference between men and women over all codes, we can use Fisher's method, by summing up the log of all p-values, and comparing to a chi square distribution with 68 degrees of freedom (twice the number of hypotheses). Note that  $-2 \sum_{i=1}^{34} \log \alpha_i = 59.61$ .

We thus get a p-value of 0.25, which is of course not significant.

For the non-binary variables we plotted their histograms and used them to partition the range into two. For example, for the age variable the age 30 was a suitable threshold leading to a distinction between subjects that were less than 30 years old and those who were 30 or over. Performing the same analysis



**Table 6** Results of fitting a generalized linear model to estimate the effect of demographic factors.

<i>factor</i>	<i>value</i>	<i>stderr</i>	<i>DF</i>	<i>t-value</i>	<i>p-value</i>
$\mu$ (intercept)	9.97835	0.07956	168.07	125.419	0.0000
$\beta_1$ (female)	-0.01930	0.05789	155.15	-0.333	0.7393
$\beta_2$ (age $\geq$ 30)	0.16118	0.07073	162.22	2.279	0.0240*
$\beta_3$ (experience $\geq$ 5)	-0.15532	0.08133	164.77	-1.910	0.0579
$\beta_4$ (assess $\geq$ 4)	-0.09124	0.06444	160.20	-1.416	0.1587
$\beta_5$ (degree)	-0.03264	0.05602	159.71	-0.583	0.5610

on this and other demographic variables produced essentially the same results as for sex: none of them led to statistically significant differences.

Finally, we also fit a generalized linear model to the data as described in Section 6.3.5, and considered the coefficients of the different variables. The results are shown in Table 6. Residual and QQ-plots indicate that normality and homoskedasticity assumptions approximately hold. Note that the value of  $\mu$  represents the average of the logarithm of the time to answer a question in milliseconds; for coefficients it is the fraction of change each variable induces (and multiplying them by 100 would give the change in percents). The degrees of freedom are approximated.

The coefficients for all the variables but one were not statistically significant. The one significant factor was age, where age 30 and above added about 16.1% on average to the time to correct answer. Years of experience above 5 led to a reduction of 15.5% on average, but this was not statistically significant. Nevertheless, this combination is interesting because age and years of experience are somewhat correlated, with a Pearson correlation coefficient of 0.67. Indeed, a potential problem with linear models like this one occurs when explanatory variables are correlated to each other. We therefore also checked partial models where one of the variables is excluded. Excluding age led to a model with no statistically significant effects; in particular, experience was also much farther from being significant. Excluding experience led to a model where self assessment was the only statistically significant effect, and age was not. These results may indicate that the correct interpretation is not that age 30 and above adds to time to correct answer, but rather that *inexperienced subjects* of age 30 and above need more time. For experienced subjects, on the other hand, the experience compensates for the age. In the original model both effects exist, and one of them is statistically significant. But when one of the variables is excluded, the model can't separate the effects of age and experience, so they counteract each other and no effect is seen. Adding an interaction variable of age and experience does not change the significance results either, because the effect is due to the combination of age *without* experience.

An additional observation is that the standard deviation of the random variable  $\zeta_i$  (which represents the experimental subjects) is about 0.1389, while the standard deviation of  $\eta_j$ , that accounts for question type, is about 0.2725. Adding the question serial number as a variable, which accounts for learning

effects as shown in Figure 5, does not change much the significance results of the variables presented in Table 6.

To summarize, our results show a general lack of significant and meaningful effects due to demographic variables, with only one test showing one statistically significant but not very meaningful effect (a reduction of 4 seconds from 23). Comparing with previous literature, we find that some prior work also did not find demographic effects, while some studies did. The most extensive literature concerns the differences between experienced programmers and novices (or between advanced students and freshmen). The majority of results show that more experienced programmers work better, typically taking less time and using higher levels of abstraction (see Feitelson for an extensive survey [26]). But interestingly, there have also been a number of studies showing the opposite. One possible interpretation is that experience leads to awareness of potential problems, so experienced programmers take more time to make sure they are correct [12]. Another is that experience leads to expectations, and if these are violated it may lead to more mistakes [33]. In our case a possible explanation for the lack of difference is that the experimental task (understanding short basic code snippets) is not challenging enough to bring out differences between programmers with different levels of experience. Also, we did not design the experiment specifically to investigate the effect of experience, and did not select subjects so as to emphasize differences in experience.

There have been very few studies on the interaction of sex with programming in general and with program comprehension in particular. Gramß et al. found that female mechanical engineering students did not do as well as their male counterparts in software engineering tasks [30]. Closer to our focus, Sharafi et al. found that fixation patterns of women reading code are different from those of men [67]. Our study differs in using professionals rather than students, and probably using easier code.

## 9 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 and the correctness of the answer, which are both unambiguous. However, note that these variables are just a proxy for “understanding”. And our underlying interest is in the effect of code complexity on understanding. One can question whether the time and accuracy of providing the output of a code snippet really reflect understanding, and indeed our own results indicate that making errors may reflect mismatched expectations rather than the complexity of the code. We leave the deeper discussion of what exactly is meant by “understanding” and how to measure it to future work. At the same time we note that at present there are no good alternatives to our methodology of measuring time and

correctness on code-related tasks, and that this methodology is universally used in the code comprehension community [61].

*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 or some other metric, which may have an effect. In particular, we identify the following threats to internal validity.

- Perhaps the most prominent 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 [36,28]. Our experiments are subject to this threat because snippets vary considerably in length. To counter this threat, we could in principle inflate shorter snippets by adding meaningless lines (e.g. `var z = 1;`) until all snippets reach the same length. However, this would most probably lead to worse confounding effects as experimental subject struggle to figure out why such redundant code is there. The conclusion is then that variations in length are an inherent property of our methodology, and we have to compromise on this issue. In retrospect, however, we can say that the results indicate that this concern is overrated. The biggest differences in length occur between the “logic” snippets (a), (b), and (c)) and the “structure” snippets (as, bs, and cs). As reported in Table 3 the differences within each pair of snippets was not statistically significant. Only when the number of observations was increased by pooling all of them together did the results achieve statistical significance at the 0.05 level, but not after Bonferroni correction. Moreover, the difference was small (less than 2 seconds), and contrary to expectation: the set of longer snippets took less time! Thus it appears that length per se is not a significant confounding factor in our experiments.
- A related issue is that the value of `x` (the number that is tested for inclusion in the given number ranges) affects the part of the code snippet that has to be looked at. For example, if `x` is in the first range in a simple disjunction, you do not have to look at the following disjuncts and thus save time. To avoid such effects we made sure the selected values indeed require the whole expression to be traversed, so short-circuiting would be impossible.
- A third concern is that the different snippets have different outputs. In particular, the snippets based on a tree of simple `ifs` provide more information regarding the location of `x` than those using a single compound logic expression. Like the length issue cited above, this is an inherent difference that we must compromise on, but it appears to be insignificant in practice.
- In a related vein, in flat compound statements the ends of the number ranges can appear in numerical order, but when using nested `ifs` the order must be manipulated in correspondence with the structure. For example, in the recursive style it is necessary to start from the middle (snippet c) as opposed to a), and such a lack of order may be expected to lead to more

difficulties. Again, this problem is inherent and indeed leads to a potential threat. However, we note again that the results did not show significant differences.

- Our results indicate that idiomatic code is somewhat easier than code that violates idioms. At a higher level, code that is easily recognized as implementing a well-known programming plan (e.g. a linear search in an array) can be expected to be much easier than syntactically similar code that does not. We avoid this potential issue by not using snippets that correspond to well-known plans.
- Finally, 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. However, we use the same style for all snippets, so this is not expected to cause a confounding effect.

Another problem potentially leading to a threat to internal validity is the possibility of learning effects. Since most of the snippets actually perform the same logic, and some of them are very similar to each other, there is a threat of a learning effect with the progress from one question to the next. This is mitigated by two means. First, during the experiment, we randomize the order in which snippets are presented. Consequently the results for each snippet are a mix of results from different locations in the sequence, and we avoid a systematic bias. Second, during the analysis, we checked the effect of adjusting the measured times to factor out the observed trend of shorter times to successive snippets. This eliminates learning as a source of variability and may be expected to improve experimental power. However, we found that such adjustments do not have a significant effect. We also note that observed subjects did not notice the commonality of the snippets.

Two additional threats concern the experimental subjects and their behavior. 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 are interpersonal differences between subjects. One well-known type of difference is in their capabilities. Another aspect 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: some chose to check themselves, while others answered immediately when they thought they were correct and moved on.

Both these threats are mitigated by randomization and by using within-subject analysis.

*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 find-

**Table 7** Main results for the different research questions

<i>RQ</i>	<i>description</i>	<i>results</i>
1	<b>if</b> vs. <b>for</b>	<b>for</b> loops are harder than <b>ifs</b>
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 abnormal loops lead to more errors
4	time vs. correctness	not necessarily correlated errors may reflect misconceptions
5	learning effect	small learning effect main improvement due to dropouts
6	demographic variables	only age possibly had a statistically significant effect

ing 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.

## 10 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. The more time it takes to interpret a code snippet, and the more mistakes that are made in the process, the harder the snippet is considered. We used this to measure the performance of 220 professional programmers as they interpret up to 15 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 `ifs`. 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 7.

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 cases`, 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 [40].

To start with, we are already planning additional experiments that focus on different styles of negation and using 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 the gamification 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 between interpretation and comprehension. Brooks makes a distinction between the essence of a software system, which is “a construct of interlocking concepts”, and its representation in code [14]. Real comprehension involves a reconstruction of the conceptual construct from the representation. Our work is at the level of deciphering the representation, namely the code. We believe that there is still a lot to be learned regarding how we read and understand code, and that this is a prerequisite for meaningful studies of deeper comprehension. Specifically, a better appreciation of the factors that affect the interpretation of code is a first step in addressing the deeper issues of what affects understanding and how to aid comprehension. At the same time, such an appreciation can also lead to practical benefits, for example by guiding programming conventions and tools.

## 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, and to the anonymous reviewers for their comments and suggestions.

## References

1. S. Abrahão, C. Gravino, E. Insfran, G. Scanniello, and G. Tortora, “Assessing the effectiveness of sequence diagrams in the comprehension of functional requirements: Results from a family of five experiments”. *IEEE Trans. Softw. Eng.* **39**(3), pp. 327–342, Mar 2013, DOI: 10.1109/TSE.2012.27.
2. B. Adelson and E. Soloway, “The role of domain experience in software design”. *IEEE Trans. Softw. Eng.* **SE-11**(11), pp. 1351–1360, Nov 1985, DOI: 10.1109/TSE.1985.231883.
3. A. Agresti and M. Kateri, *Categorical Data Analysis*. Springer, 2011.
4. S. Ajami, Y. Woodbridge, and D. G. Feitelson, “Syntax, predicates, idioms — what really affects code complexity?” In *25th Intl. Conf. Program Comprehension*, pp. 66–76, May 2017, DOI: 10.1109/ICPC.2017.39.
5. 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.
6. 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.
7. E. Avidan and D. G. Feitelson, “Effects of variable names on comprehension: An empirical study”. In *25th Intl. Conf. Program Comprehension*, pp. 55–65, May 2017, DOI: 10.1109/ICPC.2017.27.
8. 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.
9. R. Bednarik and M. Tukiainen, “An eye-tracking methodology for characterizing program comprehension processes”. In *4th Symp. Eye Tracking Res. & App.*, pp. 125–132, Mar 2006, DOI: 10.1145/1117309.1117356.
10. G. R. Bergersen and J.-E. Gustafsson, “Programming skill, knowledge, and working memory among professional software developers from an investment theory perspective”. *J. Individual Differences* **32**(4), pp. 201–209, Nov 2011, DOI: 10.1027/1614-0001/a000052.
11. 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.
12. B. Bishop and K. McDaid, “Spreadsheet debugging behaviour of expert and novice end-users”. In *4th Intl. Workshop End-User Software Engineering*, pp. 56–60, May 2008, DOI: 10.1145/1370847.1370860.
13. J. Bishop, R. N. Horspool, T. Xie, N. Tillmann, and J. de Halleux, “Code hunt: Experience with coding contests at scale”. In *37th Intl. Conf. Softw. Eng.*, vol. 2, pp. 398–407, May 2015, DOI: 10.1109/ICSE.2015.172.
14. F. P. Brooks, Jr., “No silver bullet: Essence and accidents of software engineering”. *Computer* **20**(4), pp. 10–19, Apr 1987, DOI: 10.1109/MC.1987.1663532.

15. 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.
16. 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.
17. 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.
18. R. Coe, "It's the effect size, stupid: What effect size is and why it is important". In *Conf. British Educational Research Assoc.*, Sep 2002.
19. B. Curtis, "Substantiating programmer variability". *Proc. IEEE* **69(7)**, p. 846, Jul 1981, DOI: 10.1109/PROC.1981.12088.
20. 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.
21. 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.
22. 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.
23. 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.
24. E. W. Dijkstra, "Go To statement considered harmful". *Comm. ACM* **11(3)**, pp. 147–148, Mar 1968, DOI: 10.1145/362929.362947.
25. 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.
26. D. G. Feitelson, "Using students as experimental subjects in software engineering research – a review and discussion of the evidence", Dec 2015. ArXiv:1512.08409 [cs.SE].
27. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
28. Y. Gil and G. Lalouche, "On the correlation between size and metric validity". *Empirical Softw. Eng.* **22(5)**, pp. 2585–2611, Oct 2017, DOI: 10.1007/s10664-017-9513-5.
29. 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.
30. D. Gramß, T. Frank, S. Rehberger, and B. Vogel-Heuser, "Female characteristics and requirements in software engineering in mechanical engineering". In *Intl. Conf. Interactive Collaborative Learning*, pp. 272–279, Dec 2014, DOI: 10.1109/ICL.2014.7017783.
31. 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.
32. J. Hamari, D. J. Shernoff, E. Rowe, B. Collier, J. Asbell-Clarke, and T. Edwards, "Challenging games help students learn: An empirical study on engagement, flow and immersion in game-based learning". *Comput. Human Behavior* **54**, pp. 170–179, Jan 2016, DOI: 10.1016/j.chb.2015.07.045.
33. M. Hansen, R. L. Goldstone, and A. Lumsdaine, "What makes code hard to understand?", Apr 2013. ArXiv:1304.5257v2 [cs.SE].
34. A. Heathcote, S. Brown, and D. J. K. Mewhort, "The power law repealed: The case for an exponential law of practice". *Psychonomic Bulletin & Review* **7(2)**, pp. 185–207, Jun 2000, DOI: 10.3758/BF03212979.
35. 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.
36. 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.
37. 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.



38. 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.
39. 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.
40. A. Jbara and D. G. Feitelson, “How programmers read regular code: A controlled experiment using eye tracking”. *Empirical Softw. Eng.* **22(3)**, pp. 1440–1477, Jun 2017, DOI: 10.1007/s10664-016-9477-x.
41. H. Kahney, “What do novice programmers know about recursion”. In *SIGCHI Conf. Human Factors in Comput. Syst.*, pp. 235–239, Dec 1983, DOI: 10.1145/800045.801618.
42. 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.
43. K. Kirkpatrick, “Coding as sport”. *Comm. ACM* **59(5)**, pp. 32–33, May 2016, DOI: 10.1145/289867.
44. 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.
45. D. Landman, A. Serebrenik, and J. Vinju, “Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods”. In *Intl. Conf. Softw. Maintenance & Evolution*, Sep 2014.
46. 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.
47. T. Lumley, P. Diehr, S. Emerson, and L. Chen, “The importance of the normality assumption in large public health data sets”. *Annual review of public health* **23(1)**, pp. 151–169, 2002.
48. 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.
49. T. McCabe, “A complexity measure”. *IEEE Trans. Softw. Eng.* **SE-2(4)**, pp. 308–320, Dec 1976, DOI: 10.1109/TSE.1976.233837.
50. 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.
51. 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.
52. R. H. Myers, D. C. Montgomery, G. G. Vining, and T. J. Robinson, *Generalized Linear Models: With Applications in Engineering and the Sciences*. John Wiley & Sons, 2010.
53. 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.
54. A. Newell and P. S. Rosenbloom, “Mechanisms of skill acquisition and the law of practice”. In *Cognitive Skills and Their Acquisition*, J. R. Anderson (ed.), pp. 1–55, Lawrence Erlbaum Assoc., 1981.
55. W. Z. Nunez, V. J. Marin, and C. R. Rivero, “ARCC: Assistant for repetitive code comprehension”. In *11th Joint European Softw. Eng. Conf. & Symp. Foundations of Softw. Eng.*, pp. 999–1003, Sep 2017, DOI: 10.1145/3106237.3122824.
56. 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.
57. C. Parnin, J. Siegmund, and N. Peitek, “On the nature of programmer expertise”. In *28th Psychology of Programming Interest Group Ann. Workshop*, Jul 2017.
58. D. H. Pink, *Drive: The Surprising Truth About What Motivates Us*. Riverhead Hardcover, 2009.
59. P. Piwowarski, “A nesting level complexity measure”. *SIGPLAN Notices* **17(9)**, pp. 44–50, Sep 1982, DOI: 10.1145/947955.947960.
60. L. Prechelt, “Comparing Java vs. C/C++ efficiency differences to interpersonal differences”. *Comm. ACM* **42(10)**, pp. 109–112, Oct 1999, DOI: 10.1145/317665.317683.

61. V. Rajlich and G. S. Cowan, "Towards standard for experiments in program comprehension". In 5th *IEEE Intl. Workshop Program Comprehension*, pp. 160–161, Mar 1997, DOI: 10.1109/WPC.1997.601284.
62. C. Rich, *Inspection Methods in Programming: Clichés and Plans*. A.I. Memo 1005, MIT Artificial Intelligence Laboratory, Dec 1987.
63. 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.
64. 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.
65. 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.
66. 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.
67. Z. Sharafi, Z. Soh, Y.-G. Guéhéneuc, and G. Antoniol, "Women and men — different but equal: On the impact of identifier style on source code reading". In 20th *Intl. Conf. Program Comprehension*, pp. 27–36, Jun 2012, DOI: 10.1109/ICPC.2012.6240505.
68. 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.
69. 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.
70. J. Siegmund, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring and modeling programming experience". *Empirical Softw. Eng.* **19**(5), pp. 1299–1334, Oct 2014, DOI: 10.1007/s10664-013-9286-4.
71. J. Siegmund and J. Schumann, "Confounding parameters on program comprehension: A literature survey". *Empirical Softw. Eng.* **20**(4), pp. 1159–1192, Aug 2015, DOI: 10.1007/s10664-014-9318-8.
72. 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.
73. S. Sonnentag, "Expertise in professional software design: A process study". *J. App. Psychol.* **83**(5), pp. 703–715, Oct 1998, DOI: 10.1037/0021-9010.83.5.703.
74. S. Sonnentag, C. Niessen, and J. Volmer, "Expertise in software design". In *The Cambridge Handbook of Expertise and Expert Performance*, K. A. Ericsson, N. Charness, P. J. Feltovich, and R. R. Hoffman (eds.), pp. 373–387, Cambridge University Press, 2006.
75. 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.
76. 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.
77. B. L. Welch, "The significance of the difference between two means when the population variances are unequal". *Biometrika* **29**(3/4), pp. 350–362, 1938.
78. E. J. Weyuker, "Evaluating software complexity measures". *IEEE Trans. Softw. Eng.* **14**(9), pp. 1357–1365, Sep 1988, DOI: 10.1109/32.6178.
79. K. J. Yoder and M. K. Belmonte, "Combining computer game-based behavioral experiments with high-density EEG and infrared gaze tracking". *J. Vis. Exp.* **46**, art. no. e2320, Dec 2010, DOI: 10.3791/2320.