**Noname manuscript No.**
(will be inserted by the editor)

# Considerations and Pitfalls for Reducing Threats to the Validity of Controlled Experiments on Code Comprehension

**Dror G. Feitelson**

Received: date / Accepted: date

**Abstract** Understanding program code is a complicated endeavor. As a result, studying code comprehension is also hard. The prevailing approach for such studies is to use controlled experiments, where the difference between treatments sheds light on factors which affect comprehension. But it is hard to conduct controlled experiments with human developers, and we also need to find a way to operationalize what "comprehension" actually means. In addition, myriad different factors can influence the outcome, and seemingly small nuances may be detrimental to the study's validity. In order to promote the development and use of sound experimental methodology, we discuss both considerations which need to be applied and potential problems that might occur, with regard to the experimental subjects, the code they work on, the tasks they are asked to perform, and the metrics for their performance. A common thread is that decisions that were taken in an effort to avoid one threat to validity may pose a larger threat than the one they removed.

**Keywords** Controlled experiment · Code comprehension · Experimental methodology · Threats to validity

## 1 Introduction

Code comprehension is a major element of software development. According to Robert Martin, developers read 10 times more code than they write [87].

Dror Feitelson holds the Berthold Badler chair in Computer Science.
This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 832/18).
This paper is an extended version of an "honorable mention" paper from ICPC 2021.

Authors' affiliation:
The Rachel and Selim Benin School of Computer Science and Engineering
The Hebrew University of Jerusalem, 91904 Jerusalem, Israel
Tel.: +972-2-5494555
E-mail: feit@cs.huji.ac.il

In one survey, 95% of developers said understanding code was an important part of their job, and a large majority said they do it every day [32]. Most developers also agree that understanding code written by others is hard. As researchers, we are interested in exactly what makes it hard, and what can be done about it.

Controlled experiments are at the heart of research on code comprehension [157,131]. In such experiments the participants are asked to perform a programming task based on some code. The task is crafted so that performing it successfully requires the code to be understood. By measuring the effort and success of performing the task, one can therefore obtain some information on the difficulty of understanding the code. Repeating the measurements on modified code or using various tools then sheds light on the effect of code features and tools on program comprehension.

While the general framework of code comprehension experiments is well known, there are many variations in the details. This is a natural result of the combination of the many decisions that have to be taken:

- One has to select the code on which the subjects will work. The code often reflects the nature of the study, e.g. using a certain style of identifiers if the effect of such styles on comprehension is the focus of the study. However, many other attributes of the code may also affect the comprehension process. It is therefore important to select code that does not introduce threats to the validity of the study.
- One has to select the tasks to be performed. The tasks are supposed to require understanding, but what does "understanding" mean? A particular risk is that subjects may be able to find shortcuts and perform the task without actually understanding the code, thereby undermining the whole experiment.
- One has to select the metrics by which performance will be measured. Different metrics may actually measure different things, and reflect different aspects of the difficulties in understanding code — or some factor that is unrelated to understanding the code.
- One also has to select the subjects themselves. A much-cited threat is the use of students as experimental subjects. But when are students indeed a problem, and when can they be used safely? And is the student/professional dichotomy indeed the correct one to be concerned about?

Most work on the methodology of empirical software engineering focuses on experimental design, statistical tests, and reporting guidelines [8,74,159,130,73,135]. But it is also important to get the domain-specific core features right [23]. For program comprehension our focus will be on general attributes of the code, the tasks, and the measurements. We will discuss the manipulations that are part of a specific research agenda only so far as they interact with such attributes. Our goal is to review the choices that have been made in experimental studies, and the threats to validity involved in them. This should be considered as a basis for discussion, not a comprehensive listing. Hopefully

this will encourage additional work on the methodological aspects of code comprehension research.

Some previous work in this domain includes the following. In an early work Brooks identified four factors which affect comprehension: what the program does, the program text, the programmer's task, and individual differences [22]. These foreshadow some of our observations on the code, the task, and the experimental subjects. Littman et al. defined understanding at a somewhat higher level of abstraction [84]. According to them, understanding a program comprises knowing the objects the program manipulates and the actions it performs, as well as its functional components and the causal interactions between them. Note that this description relates to systems, and not to smaller elements of code.

To the best of our knowledge the only empirical evaluation of methods to measure code comprehension was conducted by Dunsmore and Roper more than 20 years ago [44]. Their conclusion was that tasks performed using mental simulation of the code provided the best results, and that a simple question regarding perceived comprehension was also a good indicator. However, these results are of only a preliminary nature.

Siegmund et al. report on their experiences with conducting controlled experiments on program comprehension, placing an emphasis on the need to control for programming experience [132]. Perhaps the closest to our work is Siegmund and Schumann's review of confounding parameters in program comprehension [137]. This includes a catalog of 39 factors that may influence the results of program comprehension studies. Many of them have parallels in our discussion. However, we place greater focus on the considerations involved in the technical aspects of the experiment, such as the code used and the tasks performed: for example, Siegmund and Schumann spend only one paragraph on the task, saying it may be a confounding factor, while we devote a whole section to considerations in selecting a task and how this interacts with levels of understanding the code. At the same time, many of the factors identified by Siegmund and Schumann, especially considerations involving the experimental subjects, are not repeated here. As a result the two papers largely complement each other. Another close paper is Oliveira et al. [98]. This paper presents a literature survey of code readability and understanding, with an emphasis on the tasks performed and the metrics used to assess understanding. It then relates them to a taxonomy of learning. Our focus is narrower: we perform an in-depth analysis of the factors involved in only the "understanding" level of the taxonomy, and on concrete activities performed by developers.

Finally, von Mayrhauser and Vans [152] and Storey [149] emphasize the theoretical underpinnings of program comprehension research. The most-often cited cognitive models of code comprehension are the top-down model [22] and the bottom-up model [129]. Another distinction is between the systematic strategy and the as-needed strategy [84]. While this is obviously important and worthy, our work is focused on the more technical aspects of making the experimental observations in the first place.

The following sections review issues related to the code, the task, the metrics, and the experimental subjects. In each the pertinent considerations are listed first, and then the potential pitfalls. This paper is an extended version of a paper from the 29th *International Conference on Program Comprehension* [53]. The extensions added to this version fall into two categories. First, the discussions of many of the points made throughout the paper have been fleshed out. The original conference version naturally suffered from space limitations, while in the present version it was possible to present the arguments more fully and give more examples. In addition, several figures were added. Second, a few considerations and pitfalls that were completely missing in the original version have been added. A checklist summarizing the main points that need to be attended to in conducting a code comprehension study has also been added.

## 2 The Code

Experiments on code comprehension necessarily start with code. But finding suitable code is not easy. Things to consider are the scope of the code, its level of difficulty so it will be challenging enough but not too hard for an experiment, and whether to use real code or write code specifically for the experiment. Pitfalls include the danger of misleading code on the one hand, or code that will give the task away on the other hand, including the danger of using well-known code that may be recognized, and problems with obfuscating variable and function names and how the code is presented.

### 2.1 Considerations

#### *2.1.1 Code Scope*

A central question regarding the code to use in a program comprehension study is how much code to use. There is a wide spectrum of options: a short snippet of a few lines, a method, a complete class, a package, or a full system.

The main consideration in favor of a limited scope is in cases where such a scope corresponds to the focus of the study. For example, when investigating the effect of the names of parameters on the understanding of a function, it is natural to use complete functions [6]. If investigating control structures, focused snippets containing a single program element reduce confounding effects. For example, this was done by Ajami et al. in a study that found differences in understanding loops that count up and loops that count down [4]. An additional consideration is that a limited scope allows for a manageable experiment, for example not extending beyond a single hour of the experimental subject's time and sometimes as short as 10 or 15 minutes.

Scope is also an important confounding factor [59]. If you want to compare two constructs, and one requires more lines of code than the other, should

differences in performance be interpreted as resulting from the constructs or from the amount of code involved? This has no good solution, as artificially padding the shorter version may compromise the integrity of the code and cause a confounding effect worse than the difference in length. However, if the longer code turns out to be beneficial, this strengthens the results [71].

If the focus is on understanding as it is done during real development, e.g. to fix a reported bug, a large volume of code should be used. Ideally, the whole system should be available, just as it would be in a real-world setting [1]. This is important since understanding a full system is quite different from understanding a limited amount of code [82]. Brooks suggests that this difficulty is due to the software's myriad possible states [21].

In the past something that passes for a full system could involve relatively little code, thereby enabling practical experiments on "complete systems". For example, in the mid 1980s Littman et al. used a 250-line, 14-subroutine Fortran program that maintains a database of personnel records. Today such a volume more realistically represents a single class. As a result, experiments often make compromises. For example, a bug fixing task may skip the stage of locating the relevant code in the system, and focus only on the actual fix of the function in which the bug occurs.

The alternative is to conduct large scale experiments. For example, Wilson et al. asked graduate students to implement change requests in programs comprising about 100 Kloc, 800 classes, and 500 files [158]. Sjøberg et al. suggest that realistic experiments should be based on hiring professional programmers for relatively long periods of days to months [139,140]. In such a setting, subjects can work in a realistic environment, including having access to all the relevant code. This is important because comprehension—like development—is an incremental process. It takes time and accumulates, and short experiments cannot evaluate this. At best they can focus on a well-defined single step.

Another alternative is to observe professionals during their work. This approach was taken by von Mayrhauser and Vans, who analyzed maintenance sessions of professionals working on large-scale full systems. For example, in one paper they report in detail on a 2-hour session devoted to porting client programs to a new operating system [153], and in another they report on two 2-hour sessions, one fixing a bug and the other searching for the location to insert new code [155].

### *2.1.2 Code Difficulty*

Probably the most important characteristic of code used in an experiment is that the code should be appropriate for the task and the subjects. It should not be too easy and not too hard.

As an example of easy code, consider Fig. 1 (Listing 1 from Busjahn et al. [26]). This listing comprises 22 lines of code. It defines a class Vehicle with a constructor and a method, followed by a main function that creates a Vehicle object and calls the method. The method increments the vehicle's speed, subject to not going over a maximal value. So essentially all this code

```
public class Vehicle {
    String producer, type;
    int topSpeed, currentSpeed;
    public Vehicle(String p, String t, int tp) {
        this.producer = p;
        this.type = t;
        this.topSpeed = tp;
        this.currentSpeed = 0;
    }
    public int accelerate(int kmh) {
        if ((this.currentSpeed + kmh) > this.topSpeed) {
            this.currentSpeed = this.topSpeed;
        } else {
            this.currentSpeed = this.currentSpeed + kmh;
        }
        return this.currentSpeed;
    }
    public static void main (String args[ ]) {
        Vehicle v = new Vehicle("Audi","A6",200);
        v.accelerate(10);
    }
}
```

**Fig. 1** *Example of trivial code.* (©2015 IEEE. Reprinted, with permission, from Busjahn et al. [26])

does is to increment an integer. Whether this is a problem depends on its use in the experiment. The original experiment was to use an eye tracker to follow the code reading order, so very simple code is a suitable base case. But using such code in a comprehension study would probably actually measure the ability to find the one line that does something.

As an example of hard code consider Fig. 2 (Figure 3 of Beniamini et al. [14]). This is 11 lines long, comprising an initialized array and a function. The array is a lookup table. The function calculates the number of 1 bits in an input buffer by using the top and bottom halves of each byte as indexes to the lookup table and summing. While short, this code is non-trivial due to the use of bitwise operations to manipulate array accesses.

The issue of difficulty is closely associated with code complexity. Unfortunately the term "complexity" is used in three different meanings in the context of software:

– The simplest, which we emphasize here, is *code* complexity. This is a direct property of the code—as text which expresses a set of instructions—that makes it hard to understand. For example, McCabe suggested that the number of branch points in a function is a measure of such complexity [88], and Dijkstra claimed that goto statements are especially harmful [41].

```
int32_t num_set_bits[] = { 0, 1, 1, 2,
                           1, 2, 2, 3,
                           1, 2, 2, 3,
                           2, 3, 3, 4 };
uint32_t f(uint8_t* data, int data_length) {
    uint32_t count = 0;
    for (int i=0; i<data_length; i++)
        count += num_set_bits[data[i] & 0x0F] +
                 num_set_bits[(data[i] >> 4) & 0x0F];
    return count;
}
```

**Fig. 2** *Example of difficult code using bit operations to index an array.* (©2017 IEEE. Reprinted, with permission, from Beniamini et al. [14])

However, more recent research has questioned whether such code metrics indeed predict comprehension difficulty [40,94,51,93,59].

- A completely different issue is the *conceptual* complexity of the software. This is what Brooks calls "essential complexity", and is what makes software development hard [21]. Naturally it may also affect code comprehension studies.
- the third meaning is *algorithmic* or *computational* complexity, as in the number of steps required to perform some computation. This is unrelated to our interests here.

As noted above, code difficulty may be related to the use of specific code constructs — such as `goto` or bitwise operations. Pointers have also been observed to be hard to master [100,147]. More generally, code smells and anti-patterns can make code harder to understand [127,1,106]. All these obviously justify being studied, but should probably be avoided when they are not the focus of the study.

Assessing whether code is of suitable difficulty is hard, because this issue interacts with the subjects. For example, if subjects don't know about bitwise operations, code using such operations becomes impenetrable. A similar problem occurs when understanding the code requires specific domain knowledge. This needs to be checked in pilot studies and during subject recruitment.

### *2.1.3 Code Source*

The considerations regarding what code to use depend on the type of experiment being conducted. When style or tools are being investigated, the code should be "representative" of code in general. However, given the vast amount of code that exists—much of which is closed source—it is unrealistic to try to create a statistically representative sample of code. But we can at least use some sample of real code.

Volumes of real code are now freely available in open source repositories. An unanswered question is whether this is also representative of proprietary code. There are dissenting opinions on which approach produces better code (and by implication, also clearer code) [103,109]. Proponents of open source cite Linus's law, and claim that open source is better due to being subject to review by multiple users [112]. Alternatively, proprietary code has been claimed to be better because it is more managed in terms of testing and documentation.

One major concern with using real code is that the code was written by people who know what it is for. So the code may rely on implicit domain knowledge or reflect unknown assumptions and constraints. If experimental subjects in code comprehension experiments lack this knowledge, they will be unable to understand the code. A possible solution is to use code from utility libraries (e.g. performing array or string operations) [6], or to otherwise ensure that domain knowledge is not required.

The quest for self-contained code may suggest the use of functions that perform some computation that is completely devoid of any context. A good source for such functions is web sites with programming exercises for job interviews, such as leetcode.com. The advantage of such sites is that hundreds of exercises are available, often with dozens of solutions for each. However, many of the problems are unrealistic, for example implementing a contrived complicated rule to distribute candy among children. It is highly unlikely that anyone would actually be required to write such code except in a coding exercise. As a result understanding the code can be difficult simply because it does not have a clear purpose and does not make sense. It is therefore recommended to vet candidate codes carefully, and use only codes for realistic problems.

In experiments focused on particular features of the code, using real code may provide only imperfect examples, and at the same time it may introduce unwanted confounding factors. It may therefore be necessary to write code snippets specifically for the experiment to better control the different treatments. For example, Ajami et al. wrote code snippets with exactly the same functionality using different programming constructs, to investigate the effect of these constructs on understanding [4]. Note, however, that different treatments can also be based on real code. Abbes et al. used 6 large systems that contained specific antipatterns in an experiment on the effect of these antipatterns on comprehension [1]. To create the alternative treatment they refactored the systems to remove the antipatterns, without changing the rest of the design.

An extreme case is using randomly-generated code [68]. This is by definition devoid of meaning, which raises the question of what we are asking subjects to understand. It can perhaps be used to study very technical aspects of reading, as a way to separate them from the effect of semantics. For example, Stefik and Siebert used randomly-generated keywords as a baseline (like a placebo) for studying the effect of syntax on understanding [148].

In any case, regardless of the source of the code, one should ensure that it compiles and runs correctly. Few things are more embarrassing than having

```
public static int[] Cnc(int[] str, int[] end)
{
    int len = str.length;
    var rsl = new int[len * 2];
    for (int idx = 0; idx < len; idx++)
    {
        int frs = str[idx];
        int scn = end[idx];
        rsl[idx] = frs;
        rsl[idx + 1] = scn;
    }
    return rsl;
}
```

**Fig. 3** *Example of mechanistic abbreviations.* (Reprinted by permission from Springer Nature from Hofmeister et al. [67], ©2019)

subjects in an experiment point out a bug (where this is not part of the experiment).

## 2.2 Pitfalls

Even when all the considerations are taken into account, problems with the code can threaten the validity of the study.

### *2.2.1 Misleading Code*

Perhaps the biggest problem is unintentionally misleading code. If the code is misleading, subjects may make mistakes not because of the studied effect but because they were misled. Gopstein et al. have identified 15 coding practices that may be misleading [60]. Examples include using an assignment as a value, using short-circuit logic for control flow (in A||B, if A is true B is not evaluated), or presenting literals in an unnatural encoding like octal. Scalabrino et al. suggest code consistency as another attribute which affects readability and understanding [117]. This refers to the consistent use of terms in variable names and in comments. In other work, Scalabrino et al. define a metric for the deceptiveness of code based on the discord between perceived comprehension and actual comprehension [116].

A major factor in misleading code appears to be names. Arnaoudova et al. call this "linguistic antipatterns", e.g. when a variable's name does not match its type, or when its plurality does not match its use [5]. A simple example appears in Fig. 3 (Listing 4 of Hofmeister et al. [67]). This includes an array named str, and a line int len = str.Length. But contrary to what might be expected, str is not a string. Rather, "str" is an abbreviation for "start", and

```
PROGRAM Purple(input, output),
VAR Max, I, Num : INTEGER,
BEGIN
    Max = 999999,
    FOR I = 1 TO 10 DO
        BEGIN
            READLN(Num),
            IF Num ☐ Max THEN Max = Num
        END
    WRITELN(Max),
END
```

**Fig. 4** *Example of misleading code. The task was to decide whether to put a < or a > in the box.* (©1984 IEEE. Reprinted, with permission, from Soloway and Ehrlich [144])

is used to denote the initial array of integers passed to a function that will change it. To further confuse matters, the array that will be appended to str is called end, a name that may be more suitable for the final result.

A striking example of the effect of misleading names was given by Avidan and Feitelson [6]. In a study about variable naming they used 6 real functions from utility libraries. Each function was presented either as it was originally written, or with variable names changed to a, b, c, etc. in order of appearance. The unexpected result was that in 3 of the functions there was no significant difference in the time to understand the different versions, and moreover, several subjects made mistakes — and all the mistakes were in the versions with the original names. The conclusion was that the names were misleading, to the degree of being worse than meaningless names like consecutive letters of the alphabet. But presumably the original developers did not intentionally choose misleading names. So the real threat is that names that look OK to the experiment designer would turn out to be misleading for the experimental subjects.

A more insidious example comes from Soloway and Ehrlich's seminal paper on programming knowledge [144]. The code samples shown in Figure 1 of that paper were meant to investigate the rule that "a variable's name should reflect its function". This was done by writing code that calculates the maximum or the minimum of a set of input numbers. The experimental subjects' task was to insert the correct relation symbol (< or >) in the expression comparing the result so far with each new input.

But in the version calculating the minimum, the code used the name max instead of min (Fig. 4). This left the initialization to a large number[1] instead of to 0 as the only clue that the code actually calculates the minimum; if one assumes that the code calculates the maximum, as the variable name suggests, such an initialization would be erroneous. In other words, the experiment

[1] They used 999999, which today looks unjustifiable; it should have been MAXINT.

did not present its subjects with a situation in which the name does not reflect its function — it presented them with a downright *contradiction*. And this contradiction pitted a central variable name against a not so prominent initialization. Subjects would have to be especially diligent to get this right.

### *2.2.2 Recognized Code*

The opposite of misleading code is easily recognized code. This may occur if textbook examples are used, e.g. a well-known sorting algorithm. Using such code may end up measuring how well-versed subjects are in the cannon of programming examples. This also applies to many of the problems available on web sites with job-interview programming exercises.

It may be tempting to nevertheless use a well-known example but modify it in some way. This is a dangerous practice, as modifying such code risks turning it into misleading code, because subjects who recognize it expect the conventional unaltered functionality. For example, altering the initialization or termination of a canonical for loop leads to a large increase in the errors made in interpreting what it does [4].

### *2.2.3 Code Structure Giveaways*

Code used in experiments should be realistic, in the sense that it could have been written in the context of a real project. Code written specifically for experiments sometimes violates this requirement. For example, it should not include parts that do not make sense, especially if such unnecessary additions may give the experiment away.

An example is shown in Fig. 5 (Figure 1 in Schankin et al. [118]). This is a class that converts variable names in under_score style to camelCase style. The class contains two helper functions, to lowercase the first letter of a word and to capitalize it. The point of the experiment is to notice that lowercase is called instead of capitalize, which is an error. But in fact there is no reason for the lowercase function to exist at all, because the class as presented supports only one-way conversion. And in the original (erroneous) code capitalize is dead code that is never called, which may provide a hint.

Another example is when the code allows the answer to be guessed. For example, Sharif and Maletic studied name recall using a multiple-choice question with the following options: fill_pathname, full_mathname, full_pathname, and full_pathnum [126]. The distractors were selected to be as similar as possible to the original name. But the first of them contains a verb, and is therefore more likely to be a function name. Two others contain unlikely word conjugations, mathname and pathnum. This leaves only one option which makes sense as a variable name, and indeed this is the correct answer.

```
public class NotationConverter {
    public String camelcase(String input) {
        String[] parts = input.split("_");
        String result = parts[0];
        for (int i = 1; i < parts.length; i++) {
            result += lowercase(parts[i]);
        }
        return result;
    }
    private String capitalize(String input) {
        String first = input.substring(0, 1);
        String other = input.substring(1);
        String converted = first.toUpperCase().concat(other);
        return converted;
    }
    private String lowercase(String input) {
        String first = input.substring(0, 1);
        String other = input.substring(1);
        String converted = first.toLowerCase().concat(other);
        return converted;
    }
}
```

**Fig. 5** *Example of problematic code. The task was to find the bug, which is that the wrong function is called.* (Republished with permission of ACM, from Schankin et al. [118], ©2018)

### *2.2.4 Problematic Code Presentation*

A potential problem in presenting code in the context of comprehension studies is what to do with comments and descriptive names. For example, header comments and method names are specifically designed to allow readers to understand what a function does without reading its code. Leaving them intact may therefore undermine an experiment where subjects are supposed to deduce just that. But given that they normally exist, removing them creates an unnatural situation. Thus it is suggested that this be done only in experiments using short code segments, where the focus justifies using an unrealistic setting. If a large body of code is used, names and comments should probably be retained. Likewise, in experiments where the task is not to understand what the code does there is no problem. An example is superficial debugging tasks, as explained below in Section 3.1.6.

As a side note, care should be taken that the comments do not directly interact with the task. For example, Figure 1 in a paper by Buse and Weimer [25] shows a short code snippet preceded by the comment "this is hard to read", from an experiment where subjects are asked to assess readability. Providing such overt hints regarding the expected answer should be avoided.

A more delicate issue is how to handle mathematical and logical expressions. For example, should one rely on operator precedence, or use parentheses to clarify the order of evaluation? Again, if this is not the issue being studied, the best approach is probably to make it as unobtrusive as possible. Any effort that subjects spend on understanding expressions, and any mistakes

they make, dilute the results that the experiment was designed to produce. In practical terms, this means to make the expressions as simple and obvious as possible, including by using parentheses.

More generally, all aspects of code readability affect its comprehension. If they are not the issue being studied (e.g. [99]), they should be controlled. When using real code it may be tempting to present the code as it was written. But if the original code is not laid out properly, or uses an idiosyncratic style, this could introduce a confounding effect. A good practice is to use an IDE's default indentation and syntax highlighting, as inconsistent presentation leads to cognitive load and therefore constitutes a confounding factor [49]. Methods within a class should be listed in calling order, meaning that called methods are placed after methods that call them [58].

Another aspect of code presentation is coding style. Fashions change, and different people write in different styles. A mismatch between the writing style and the preferences (or experience) of the experimental subject may cause bias and a confounding effect. Two things can be done to reduce such problems. First, adopt the style that matches the prevailing culture (e.g. snake_case for Python but CamelCase for Java). Second, be consistent and use the same style throughout.

#### *2.2.5 Variable Naming Side-Effects*

Variable names and function names are instrumental for comprehension, and in many cases they provide the main clues regarding what the code is about. In the context of comprehension studies this may be undesirable, so the names have to be stripped of meaning. Some of the ways that this has been done are problematic:

- Simple options are using arbitrary strings (asdf, qetmji) or unrelated words (superman, purple). These are distracting, and may be useful only in relation to extreme research questions on reading or the possible detrimental effect of extremely bad (distracting) names. They should not be used if this is not the research issue.
- Another approach is to use obfuscation, e.g. by applying a simple letter-exchange cipher [136]. This leads to names that are long and distracting non-words. For example, the function name countSameCharsAtSamePosition can change into ecoamKayiEoaikAmKayiEckqmqca. Long names like this that differ in just a couple of letters may become very hard to distinguish. Note too that obfuscation may deeply affect how subjects perform tasks. Variable names convey meaning, and thus enable a measure of top-down comprehension. Siegmund et al. therefore used obfuscated variable names to force subjects to use bottom-up comprehension based on the syntax [133, 136]. Such an effect may happen with problematic code also when this is not intentional.

Alternative better ways to obfuscate variable names are the following:

- One option is to use words that represent the technical use of the variables, but do not reveal the intent — exactly the opposite of what we usually try to do. For example, one could use num1 and num2 instead of base and exponent in a function that calculates a power [133].
- Perhaps the simplest and most straightforward approach is to just use consecutive letters of the alphabet in order of appearance [6,67]. Note that this is different from using the first letter of the "good" name, as that may still convey information [14].

A related case is when names are abbreviated to see the effect of such abbreviations. This is sometimes done in a mechanical manner in an attempt to be more scientific and reduce the reliance on individual judgment. For example a possible approach is to concatenate the first 3 consonants in the name (as shown in Fig. 3 above) [67]. However, this may lead to unnatural or misleading names, such as str for start or rsl for result. Consequently the gain in rigorosity may come at the expense of reduced validity. It is better to use judgment rather than a mechanical approach, e.g. allowing res for result and the 4-letter conc for concatenate. To reduce the danger of mistakes in judgment, the abbreviations can be derived independently by two people, and then compared.

#### *2.2.6 Inappropriate Code for the Task*

Importantly, the task subjects are required to perform and the code must be compatible. For example, when studying whether indentation aids comprehension, one needs a task that depends on the block structure of the code. Otherwise indentation is indeed not an important feature, and the results will show that it does not matter. But this would be wrong, because maybe indentation does indeed matter for another task — for example, one that is related to navigation and identification of code blocks.

For example, Miara et al. conducted a study on indentation using 102-line long code with a main and two functions, and a maximal nesting level of 3 [92]. The result was that nesting had some effect, based on questions such as whether all variables were global, which require the definition of variables at the beginning of functions to be identified (the study is from 1983 and the code was written in Pascal). Many years later Bauer et al. replicated this study, but the code used was 17-line single functions with a maximal nesting of 2, and the task was to anticipate what the program would print [10]. In this setting indentation was not found to be important, but maybe the reason was that the code structure was too simple.

## 3 The Task

If we focus on comprehension per se, experiments on code comprehension are a sort of challenge-response game. The experimenter challenges the subject to understand some code. A subject that claims to have achieved such understanding must prove it by performing some task. It is therefore vital that the

task really reflect comprehension. In a sense, the task defines what "comprehension" means. The main considerations are therefore what level of comprehension is reflected by each task. And the main pitfall is that the link between the task and the understanding might be compromised, e.g. if subjects can guess the correct answer without actually understanding the code.

## 3.1 Considerations

In real life, program comprehension is rarely an end in itself. Rather, comprehension is a prerequisite to performing some programming task, such as fixing a bug or adding a feature. So experiments can use such tasks directly.

Alternatively, one can consider comprehension itself. In this case the main consideration in selecting the task to perform is that the task reflects the level of understanding which is at the focus of the experiment. Should the subjects just know the variables and data structures? Maybe the behavior at runtime? Or perhaps also the underlying algorithms? The following subsections detail several such possible levels. We name them using the straightforward dictionary meaning of different words. Note, however, that this is not universal, and over the years some of these names and others have been used in various non-compatible ways.

It is also interesting to consider the relationship of code comprehension to reading natural language texts, and to the terminology used there. Being able to read at all depends on the *legibility* of the text — that the letters stand out clearly. This is more a matter of design (e.g. fonts and color contrast) than of reading. *Reading* is commonly referred to as combining the acts of identifying letters and words and gleaning the meaning conveyed by them. While this terminology is not universal (for example, Smith and Taffler advocate distinguishing between "reading" and "understanding" [142]), it is often also adopted in studies on reading code. For example, the first sentence in Buse and Weimer [25] is "we define readability as a human judgment of how easy a text is to understand". But code is actually somewhat different from text. For example, the difficulty of text can be approximated based on simple metrics like sentence lengths and word lengths [43,142], but at the same time text may be ambiguous (which may be used to advantage in both prose and poetry). In code such metrics are not very meaningful, and the semantics are unique and well defined. Our goal is to describe how levels of the semantics relate to levels of reading the code. The relationship between the different levels is shown in Fig. 6.

### *3.1.1 Recognition Task (tokens and structure)*

The most basic level is just recognizing the elements of the code, such as tokens and structure. Note that this has two facets. Recognizing tokens is a localized task. It is aided by programming practices such as surrounding the assignment operator with spaces, and by IDE options such as colorizing

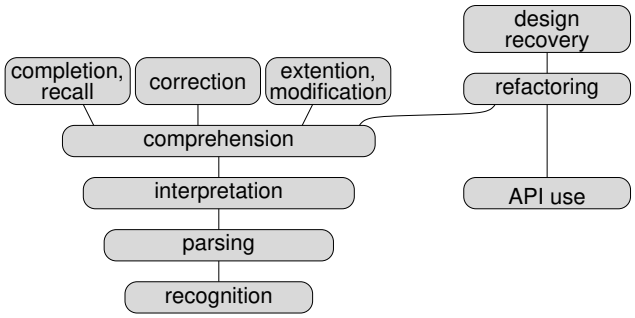


**Fig. 6** *Relationships between tasks used in comprehension experiments.*

keywords. Recognizing structure is a more global issue, which has a strong influence on navigation in the code and on the findability of key elements in it [99]. In addition to colorized keywords, this is aided by practices such as consistent indentation.

The above considerations indicate what type of tasks may be used to assess recognition. Such tasks include:

- Find a certain word, e.g. the use of a variable.
- Identify nesting of constructs, e.g. the most deeply nested one.
- Verify whether two expressions have the same syntactic structure.

But studies that focus on mere recognition are few, such as those targeting notation or syntax highlighting [99,108,68,65]. In addition, disrupted layout has been used as a control and to force subjects to employ bottom-up comprehension [136].

### *3.1.2 Parsing Task (understand syntax)*

The next level up is to be able to parse the code. This shows that you are able to understand the syntax: what are legal expressions, and what their relations may be. Example tasks can include

- Find the type of a variable (in a typed language).
- Find a syntax error in a function.
- At a larger scale, draw a basic UML class diagram of a project, or compare a UML diagram with code that is supposed to implement it. "Basic" here means without some details that require deeper understanding, such as using type inference and defining the cardinality of associations.

Note that these tasks do not require any understanding of what the code does. This is intentional, as such understanding is detailed in subsequent higher-level tasks. It is sometimes claimed that this level of understanding is not very important or interesting in itself, as syntax issues are typically delegated to a compiler. It is therefore not commonly used in comprehension experiments, and when it is, it may be used as a control, to show the difference between understanding syntax and semantics (e.g. [133,118]).

### *3.1.3 Interpretation Task (local semantics)*

While parsing requires understanding the structure of the code, interpretation requires understanding the semantics of the individual instructions. Tasks which reflect the ability to interpret code include

- Find what the code prints for a certain input. This can be done by simulating the execution one instruction at a time, much like an interpreter would.
- Answer simple questions about the code. For example, Pennington suggested using questions on the program's control flow (will the last record be counted?) and data flow (does the value of variable `a` affect that of `b`?), its states (will `c` have a certain value after the loop?), and specific operations (is `d` initialized to 0?) [105].
- Write tests that provide statement coverage or branch coverage. This only requires one to understand individual condition statements.
- Identify and remove dead code which will never be executed (e.g. a function that is not called, or a condition that is always false).
- Draw a UML sequence diagram. To do so one just needs to understand which functions call each other.

Interpretation is on the verge of "real" understanding. On the one hand, one can hand-simulate the execution of code and figure out what it will print without forming a general understanding of what the code actually does. But this is nevertheless appropriate for very short snippets of code comprising a single control block and nothing else. For example, Ajami et al. use this for comparisons of different formulations of a predicate used in an `if` construct [4].

On the other hand, there are cases where it is actually easier to figure out what the code does rather than to simulate its execution. A case in point is code with loops, especially when many iterations are performed. For example, Hannebauer et al. suggest this is the case in an experiment they perform on code that implements bubble sort [65, Figure 2].

Interpretation tasks are quite popular in comprehension experiments, because they are easy to create and to check: you just compare the given answer to the known correct answer. However, one must carefully consider the details to determine whether an answer based on tracing the execution is likely, and whether this affects the validity of the experimental results.

### *3.1.4 Comprehension Task (global semantics)*

Comprehension is understanding the underlying concepts of the code, and grasping its functionality in abstract terms. This is the general goal of code comprehension. The difference between comprehending semantics and parsing syntax is real. fMRI studies show that comprehension tasks activate different parts of the brain than syntax-related tasks — parts related to working memory, attention, and language processing [133].

The most common way to assess comprehension is to ask questions about the program. Specific questions and tasks which are thought to reflect comprehension are:

- After reading and understanding the code, answer a question about the expected output for a given input without seeing the code again — that is, without the ability to simulate its execution (this assumes the code is non-trivial and cannot be remembered easily).
- Describe the functionality of the code. More concretely, this can be achieved in several ways:
  - Ask subjects to suggest a suitable meaningful name for a function.
  - Ask subjects to summarize the purpose of the code.
  - Ask subjects to add documentation to the code, for example header comments for functions.
  - More formally, ask subjects to articulate the contract of a function or API: what are the preconditions and postconditions when using it [91].
- Describe the flow of the code, namely how it transforms its input into its output. Or more generally, perform a code summarization task. Answers to such questions must be carefully analyzed to ascertain that they indeed reflect comprehension. For example, saying that the code loops over all numbers smaller than the input and checks for cases where they divide the input number with no remainder may be precise, but it is just a technical description of the code. True comprehension is to say that the code checks whether the input number is a prime.
- Answer questions about specific elements of the code, for example the purpose of a certain variable, or why a certain function is called. Note however that such questions do not necessarily assess global understanding.
- Write a test suite for a function. As this requires the expected results to be given, it shows you know what the tested code does. A comprehensive test suite specifically shows understanding of semantic corner cases.
- Another possible task is to explain the limitations of a function or API — when should it be used, and when can't it be used. This is related to the question of writing the contracts for functions mentioned above.

It is advisable to use questions of several types, so as to cover different aspects of understanding the code [105,34]. But in many cases, assessing the answers given to comprehension tasks is not easy (as discussed in Section 4.1.2 below). Therefore other tasks which assess comprehension indirectly are sometimes used. Such options are described in the following subsections.

### *3.1.5 Code Completion or Recall Task*

A common exercise when learning foreign languages is "fill in the blanks" (the so-called cloze test): the students are given a text with some parts missing, and need to complete them either on their own or using a list of options, based on their understanding of the text and of how different options fit in. This can also be done with code [34]. For example, Soloway and Ehrlich used this to

study the effect of a mismatch between a variable's name and its function [144], and Hannebauer et al. used it to study the understanding of an inheritance hierarchy [65]. However, finding the right balance between trivial cases and misleading cases appears to be hard. As noted above, Soloway and Ehrlich's code was misleading. Hannebauer et al's is trivial: subjects were requested to replace the XXXXX in the declaration Mother x = new XXXXX() with one of the options Father, Mother, or Daughter. This can obviously be done without ever looking at the code.

A rather different type of task is to read the code, understand it, and then try to recall it from memory. Obviously this is limited to reasonably short codes, e.g. up to 20–30 lines long. The motivation for this task is the seminal work of Simon and Chase, which showed that expert chess players can easily memorize meaningful chess positions, but are not good at memorizing random placements of chess pieces [138]. Hence the memorization interacts with identification of meaning.

Shneiderman conducted an experiment based on this approach more than 40 years ago [128], concluding that better recall indeed correlates with better comprehension (as measured by the ability to make modifications to the code). McKeithen et al. also showed that expert programmers are better able to recall semantically meaningful program code [89]. However, this type of task is rather far removed from what programmers actually do, and perhaps for this reason does not seem to be popular.

#### *3.1.6 Correction Task (white-box)*

Of the different types of maintenance [83], corrective maintenance (fixing bugs) is the one most often used to test understanding. Moreover, Dunsmore et al. have found that perceived comprehension indeed correlates with finding bugs [45]. But not all bugs reflect the same level of understanding. One needs to distinguish technical bug fixing (e.g. finding and correcting a null pointer dereference [82], a method declared private instead of public [65], or a syntax error) from a semantic error (such as calling the wrong helper function [118] or using a wrong index into an array [67]). Finding technical errors is more at the level of interpretation than comprehension — it can be done by scanning the code superficially without any deep understanding of the whole. Syntax errors may be irrelevant, as they should be caught by the compiler (but nevertheless they are sometimes used, e.g. [65]). Only semantic errors reflect real comprehension.

An important question is exactly what bugs to inject. Two classifications were suggested by Basili and Selby [7]. The first is a distinction between errors of omission and errors of commission. This is an important distinction, because with commission the subjects can see the error, but for omission they need to notice that something is missing — which depends on a preconception of what the code is trying to do. The second classification lists six types: initialization, control, computation, interface, data, and cosmetic. Using such classifications helps reduce confounding effects that may be due to a specific type of bugs.

They were used for example by Juristo et al. and by Jbara and Feitelson [75, 71].

An important consideration is whether to ask only for the correction of the bug given the location where it occurs, or to also require subjects to locate the bug [104]. The first option is more focused on understanding the details of the given code. The second mixes this with achieving an overall view of how the code is structured and how responsibilities are distributed across modules and functions. This is a different level of understanding that should be assessed separately, for example by asking where to look for the bug rather than asking to fix it.

#### *3.1.7 Extension or Modification Task (large scale white-box)*

Most of the tasks outlined above are suitable for short code snippets, a function, or perhaps a class. Some of them, e.g. correction tasks, can also apply to larger software systems. Extension and modification of software can also be done on a single function, but usually the minimal relevant scope is a class, and the common scope in real-life situations is a module or a complete system.

A few examples are given by Wilson et al. who use large-scale projects of 78 and 100 KLoC to study adding new features [158]. This enables them to study not only the change itself, but also the process of finding where in the code the change must be made. If the task asks only to change a given function, it misses the steps of zeroing in on the correct location to make the change, and the evaluation of the impact that the change may have on other parts of the system [111]. Whether this is a problem depends on whether you consider it part of comprehending the system.

Importantly, writing code as in code extension or modification tasks is different from reading code as in comprehension tasks. Krueger et al. show using fMRI studies that writing activates areas in the right hemisphere of the brain, associated, inter alia, with planning and spatial cognition [80]. So it seems that in these tasks, while comprehension is needed, it is not the main activity. They may therefore be less suitable as tasks that reflect comprehension.

Moreover, there are additional levels of comprehension which may be required to correctly change code but are hard to attain and to measure. Levy and Feitelson identify two such levels beyond the usual black-box/white-box dichotomy [82]:

- "Out-of-the-box" comprehension refers to subtle interactions of the code under study with other parts of the system, e.g. as may be required for extreme optimizations.
- "Unboxable" comprehension is the appreciation of the underlying assumptions and considerations involved in developing the code, which may not be directly reflected in the code at all.

#### *3.1.8 Use Task (black-box)*

Black-box is a special case of comprehension, where we are interested in using the code as opposed to understanding how it works (white-box) [82]. This is very common and important in real life when one needs to use third-party libraries. It also forms the basis for modularity, encapsulation, and information hiding [101,102]. But it is rather uncommon and not very useful in comprehension experiments. The simplest way to exhibit black-box knowledge about an API is to use it, that is to write some code that calls the API functions. In addition, one can ask about various attributes of the API:

- Details about parameters of API functions.
- Connections between functions, e.g. if one must be called before another is called.
- Documented preconditions or constraints.

Note that black-box understanding is actually disconnected from the code itself — this is the essence of information hiding. It is based on documentation. But *generating* such knowledge requires deeper comprehension, as noted above.

#### *3.1.9 Design-Related Task (abstraction)*

Understanding a system is not the same as understanding a single module or a smaller piece of code. When understanding a system the focus is on understanding the structure, namely the system's components, what are their responsibilities, and how they interact with each other [19,82]. A deeper level of understanding is to understand why it is structured like this, that is, to understand the rationale for the design decisions that were taken during development.

Recovering the design of a system from its implementation is an act of reverse engineering [33]. One possible approach to achieve this is by analyzing the dynamic behavior of the system at runtime, and noting the interactions between its components [35]. This is different from the approaches used for other tasks listed above, which mostly focus on the static code. It may even be claimed that resorting to code execution is a way to circumvent the need to understand the code directly. However, performing tasks that affect the design do require one to contend with the code itself. Possible tasks related to understanding the design are different types of refactoring, such as the following [56]:

- Extract methods, that is identify blocks of code that should be made into independent methods for reuse or better structure.
- Suggest methods that should be moved to another class.
- Modify the inheritance hierarchy by pulling up or pushing down a field or a method, and placing them in a more appropriate class.
- Replace a conditional behavior with using inheritance to create polymorphism.
- Identify a common base-case and extract a superclass to represent it.

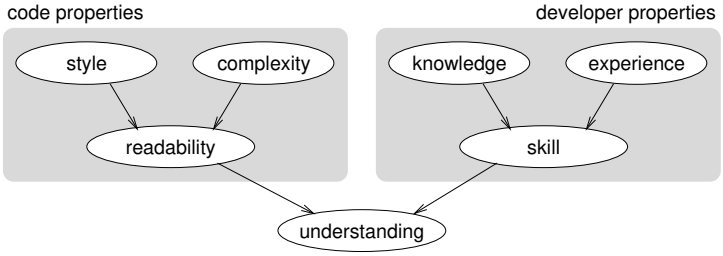


**Fig. 7** *Factors affecting the understanding of code.*

– Extract explaining variables to improve the comprehensibility of the code [30].

While there is extensive literature about the execution of such tasks, they are not common in code comprehension studies. Possible reasons are that they require a large scope to be meaningful, which is harder to provide in a controlled experiment. It is also hard to judge the correctness of performing such tasks, as design is always also partially a matter of taste.

At a higher level of abstraction, the result of the design process is an architecture. Hence understanding the design is understanding the architecture. A possible task to show this level of understanding is then to describe the architecture. This can be expressed, for example, as defined by the 4+1 views suggested by Kruchten [79]. For example, in an experiment we can ask participants to draw a conceptual diagram showing relations between entities. Note, however, that in a real-life setting comprehending a system is a continuous process, and each task adds to this understanding in an incremental manner [152]. In all likelihood such a process cannot be fully replicated in an experiment. However, design recovery may also be useful in the context of other tasks, such as debugging or adding a feature.

### *3.1.10 Selection of Tasks*

The previous subsections indicate that different tasks actually reflect different aspects of understanding. A possible way to interpret the relations between them is shown in Fig. 7. A major distinction is between factors that reflect code properties and factors that characterize the developer tasked with understanding the code. For example, style is a code property, but being able to parse and interpret code reflects knowledge of the programming language in which it is written. Thus the selection of which task to use should be predicated by what we want to study: the code or the developer. The more involved tasks, such as modifying or explaining code, typically involve both code and developer. It is then hard or impossible to claim that the task measures one or the other [18]. In addition, challenging tasks such as adding a feature conflate comprehension with other activities such as designing and programming. This dilutes the fraction of the effort invested in comprehension, thereby re-

ducing the accuracy of experiments with the express purpose of studying only comprehension.

To get a better picture it may be advisable to use multiple different tasks in the same study (as done e.g. by Pennington [105]). Multiple tasks of different types can illuminate different aspects of comprehension, and may expose unanticipated differences between treatments. With multiple tasks of the same type one can obtain more nuanced and accurate results, leading to better validity.

It would be good to also have independent assessments of the value of different tasks in measuring comprehension. Regrettably it seems that very little such research has been performed to date. One study is that by Dunsmore et al., in which they found a correlation between bug fixing and perceived comprehension [45]. But more work on this issue is required.

## 3.2 Pitfalls

### *3.2.1 Substituting Opinion for Measurement*

The main problem with selecting a task that reflects program understanding is the classic construct validity issue: are you measuring what you set out to measure? In particular, does your task actually measure understanding at the level you are interested in?

For example, consider Buse and Weimer's "A metric for software readability" [24], which—very naturally, given its title—is often cited as a reference on readability. But in the reported experiments, subjects were told to score code snippets "based on [your] estimation of readability", where "readability is [your] judgment about how easy a block of code is to understand". This reflects two problems. First, "readability" is a catch-all phrase which does not distinguish between different levels of understanding as delineated above. Second, using judgment as the dependent variable conflates personal opinion about what it means to understand (which could be any of the levels discussed above) with misconceptions about how easy or hard a specific code snippet is. This violates the whole concept of using a well-chosen and well-defined task to actually measure performance that depends on comprehension. However, one should acknowledge that many of the tasks listed above actually do not measure comprehension directly, but rather by proxy.

Scalabrino et al. face this issue head-on and distinguish between *perceived* understanding, where subjects just declare that they think they have understood a method, and *actual* understanding, where they correctly answer several verification questions [116]. However, the questions they suggested were about the meaning of a variable name, or the purpose of calling a certain function. It is debatable whether such questions indeed reflect a full understanding of the code.

### *3.2.2 The Danger of Shortcuts*

The tasks in code comprehension experiments are predicated on the assumption that they can only be performed successfully if one understands the code. But as noted above, it is not necessarily true that you need to fully understand the code to predict what it will print, or to correct a bug that it contains. Furthermore, experimental subjects may be lazy [113,82]. They may prefer to use an "as-needed" program comprehension strategy as an alternative to a "systematic" strategy leading to full understanding [84]. So given a specific task, they might make do with comprehending only whatever is directly needed for this task [154]. Unless this part is precisely aligned with the experimental objectives, this compromises the validity of the experiment.

When designing an experiment it is therefore of paramount importance to avoid tasks where the brunt of the work can be avoided. This is a significant threat to the premise that comprehension is a prerequisite for testing, debugging, and maintenance. Examples include cases where tasks can be done mechanically without understanding. For example, in bug fixing, finding a syntax error or finding a null pointer reference can be done without understanding what the function does. In code modification, a simple refactor like extracting a function can be done without understanding how the function works.

### *3.2.3 Confounding Explanations*

In controlled experiments one needs a control: a base-level treatment with which to compare the performance on the other treatments. This is what gives controlled experiments their explanatory power (which is why it is regrettable that there is such a limited use of controlled experiments [141]).

To provide explanatory power the task has to be crisp in the sense that it strongly supports a certain interpretation. Not all tasks have this property. For example, the fill in the blanks task used by Soloway and Ehrlich is not crisp, because the variable name they used is misleading (as described above) [144]. Thus a failure to answer correctly may not be due to a problem with the conceptual model (what the experiment was supposed to check), but simply due to falling in the trap of the misleading name. Likewise, failure in recalling code verbatim from memory may identify totally wrong code or code that does not abide by conventions, not necessarily hard to understand code. Finally, failure to find a bug such as calling the wrong function may be the result of lack of attention, rather than lack of understanding.

### *3.2.4 The Working Environment*

A potentially important confounding factor is the working environment in which subjects perform their task. Certain environments may include facilities that support the task and make it easier to complete. If such an environment is provided, performing the task becomes easier. Worse, having access to features

that support the task may undermine the need to understand the code or affect the process of how it is understood.

Note, however, that this also depends on the subject knowing how to use the environment. Subjects who do not know how to use the required feature (or don't know it exists) will be at a disadvantage. If some subjects know how to use these features and others do not, this becomes a confounding factor that may interfere with the results.

A possible solution to this problem is to use a reduced environment, which does not include the features that may be used to help perform the task. However, this is also problematic for subjects who are used to work in an environment which does include such support.

## 4 The Metrics

Rajlich and Cowan suggested that the dependent variables measured in comprehension studies should be the accuracy of the answers, the response time of accurate answers, and the response time of inaccurate answers [110]. Of these, the most commonly used is time to correct answer. Accuracy is also often used, especially when it is easy to assess (for example, when the task is to predict what a given code will print). The time to inaccurate answers is typically not used. An interesting question is how and whether different measurements should be combined into a single metric.

Note, however, that these metrics are actually proxies for what we are really interested in: the effort invested in understanding the code, and the difficulty of understanding the code. Recently, biophysical indicators (ranging from skin conductance through pupil size to fMRI brain activity patterns) have also been suggested as indicative of the effort expended in code comprehension. This is a potentially valuable development, but such metrics are not widely used yet.

### 4.1 Considerations

The main consideration regarding metrics is that they be measurable. This may interact with the task, as some tasks produce outcomes that are more measurable than others. Note that as discussed above we do not consider voicing an opinion as a measurement.

#### *4.1.1 Imposing Time Limits*

There are basically two approaches to measuring performance: how much one can achieve in a given time, or how long it takes to perform a given task [18]. Most experiments on comprehension measure time for a task. This is also closer to normal working conditions. However, placing a generous time limit may be advisable to exclude subjects who experience difficulties for some reason, or subjects who do not work continuously or conscientiously.

#### *4.1.2 Judging Accuracy*

If we consider comprehension experiments as a challenge-response game, the outcome of the game depends on the evaluation of the response. If the response was correct, the experimental subject has met the challenge and "wins". But how do we know whether the response was correct? This obviously depends on the details of the task.

The easy cases are when the response is well-defined in advance, such as to identify what a given code will print (e.g. [4]). In this case the answer can be checked automatically. The only reservation is that inconsequential variations (e.g. an added space) should be ignored. If multiple tasks are used, the fraction performed correctly can serve as a score.

In cases such as when the question is "what does this code do" or "give this function a meaningful name", one needs to prepare a capacity for judging the responses. This should include

- A key, prepared in advance, of what responses are expected to include, and how to identify and score each level of achievement. For example, in an experiment based on comprehension of a program that created a histogram of word occurrences in a text, a third of the points were given for answering that the program counts word occurrences, a third for saying that it prints each unique word, and a third for noting that it prints the number of occurrences next to each word [92].
- Application of the key by at least two and possibly more independent judges.
- A protocol for settling disputes, e.g. majority vote (2 of 3 judges) or conducting a joint discussion till reaching consensus.

It is also important to keep track of and report how many disagreements there were.

#### *4.1.3 Reaction to Errors*

In those cases where a wrong answer can be detected automatically, e.g. when the experimental subject is required to find out what the code will print, one has to decide what to do if a wrong answer is given. A common approach is to just go on with the experiment. Possible alternatives include

- Display a message indicating that a mistake has been made. But this may affect the rest of the experiment, either due to discouraging the subject, or due to facilitating a learning effect.
- In addition to indicating that a mistake was made, allow the subject to try again [67]. This raises the questions of how to measure time. Do you include the sum of all trials? Is it fair to compare this to the time taken by someone who did not try and fail?
- When it is expected that all subjects will succeed (which implies that correctness is not being measured), discard subjects who fail [67]. In other words, failure is used as an exclusion criterion.

#### *4.1.4 Combining Dimensions of Performance*

If both time and accuracy (correctness) are measured, the question is whether to report them separately or to combine them in some way. Combining the two metrics simplifies the analysis by making it one-dimensional. But this is justified only if they indeed reflect the same underlying concept.

Bergersen et al. suggest a crude categorical classification scheme which combines time and correctness [16]. In its simplest form, this scheme defines 3 levels of accomplishment:

1. Incorrect answer.
2. Correct answer, time above the median.
3. Correct answer, time below the median.

If the task is made up of multiple stages, the levels first reflect the number of stages completed successfully, and if all were, the time range in which this was achieved.

Beniamini et al. suggest a continuous version of such a combination, where accomplishment is defined to be the quotient of the correctness score divided by the time [14]. This can be interpreted as the "rate of answering correctly". Incorrect answers are naturally included with a rate of 0. Scalabrino et al. suggest a similar formula, but use the time saved relative to the subject who took the longest to answer [116]. This has the disadvantage that outliers may distort the results of others.

A related question is what is the significance of time to incorrect answer? The most common approach is to ignore this data. A possible alternative is to interpret such data as instances of censoring: we know that the subject spent this much time and did not arrive at a correct answer, therefore the time needed for a correct answer would be longer. Another option is to interpret this as wasted time. If the task was something practical, like fixing a bug, a wrong fix reflects waste because the task would have to be done again. A third option is to use this as an assessment of motivation: this is how much time subjects are willing to invest [110].

#### *4.1.5 Using Direct Physiological Measurements*

The commonly used dependent variables of time and accuracy measure the overall resulting performance when executing a task. But they rarely provide information about how this performance is achieved, e.g. what cognitive processes were used, and what were the trouble spots on which the experimental subjects stumbled. They also do not provide direct information on the effort needed to achieve the measured performance. In recent years there is an increasing use of tools that enable these factors to be studied too.

The most prominent tool in the context of code comprehension studies is eye trackers [120,124,97,13,123]. Eye trackers enable an identification and quantification of how the experimental subjects focus on different parts of the code, and also a recording of the gaze scan path: the order in which they

go over the code. This is especially useful to identify what the experimental subjects are interested in. An example is given by Jbara and Feitelson [72]. This study used eye tracking to quantify the amount of time spent looking at successive repetitions of the same basic structure. The results showed that the first instances get more attention, and were used to create quantitative models of how attention decreases with instance serial number.

The effort required to comprehend code can also be measured more directly than by the time and correctness of the comprehension. For example, changes in pupil size are known to be correlated with mental effort [77], and this is measured by most eye trackers. Various other biophysical indicators for effort have also been used in relation to software engineering research [57,36]. In addition, subjective self-reporting can be used, e.g. based on the NASA task load index [1].

An even deeper level of analysis of how subjects comprehend code is provided by functional magnetic resonance imaging (fMRI). This is being used in an increasing number of studies [133,136,55,69,80]. fMRI identifies areas of the brain that become active when performing a task. For example, this has enabled the distinction between brain activity patterns when performing syntactic vs. semantic tasks [133] or reading vs. writing [80]. It has also been possible to distinguish between reading code and reading prose based on brain activity patterns [55]. Finally, thinking about manipulating data structures and about spacial rotation tasks employ the same regions in the brain [122].

A technically simpler alternative is to use functional near-infrared spectroscopy (fNIRS) technology [49,122]. Unlike fMRI, which is a large noisy machine in which subjects need to lie and is expensive, fNIRS is based on wearing a scalp cap and can be done sitting in front of a computer. And it provides nearly the same level of data as fMRI.

An especially interesting attribute of neuroimaging studies is that they bypass the limitation of conscious reporting. A lot of processing in the brain is done unconsciously, and therefore subjects cannot report on precisely what they had done. Techniques such as fMRI and fNIRS provide an objective glimpse into what the brain is doing, without the need for cognizant reporting, and irrespective of potential filtering and rationalization by the conscious self.

Importantly, using all the above methodologies in the context of program comprehension studies is still pretty new. Developing the methodologies and establishing best practices is therefore an ongoing effort [12,123,122].

## 4.2 Pitfalls

### *4.2.1 Confounding Effects*

Measurements are always subject to the danger of confounding effects. Many of the pitfalls noted in the previous sections may come into play when we measure the time or accuracy of code comprehension, and lead to unreliable

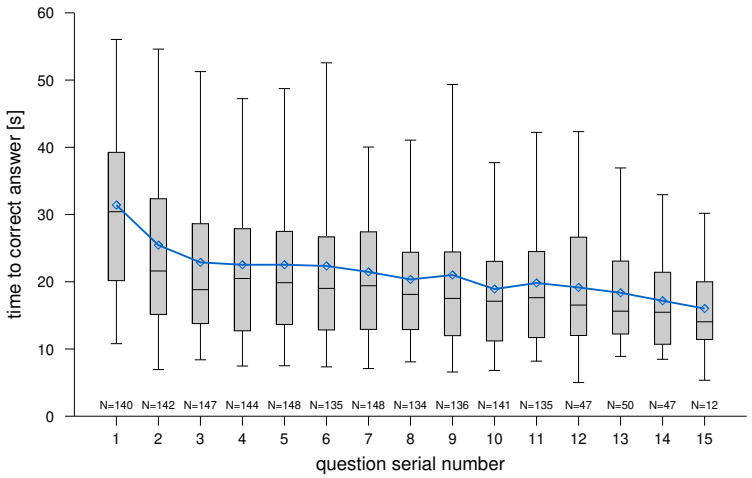


**Fig. 8** *Example of distributions of time to correct answer for a sequence of questions in an experiment.* (Reprinted by permission from Springer Nature from Ajami et al. [4], ©2019)

results — namely results which do not reflect the intended aspects of code comprehension.

One straightforward effect is getting used to the experimental setting. It is apparently not uncommon that the first task or two in a sequence take longer, as the subjects learn what exactly is required of them (e.g. Fig. 8, from Ajami et al. [4, Figure 5]). It may therefore be better to discard the first such result(s), or use them to evaluate the participants. More generally, measurements necessarily conflate the effects of code attributes with those of the person participating in the experiment (as noted above in Section 3.1.10 and Fig. 7). If the subjects are unsuitable, e.g. is they lack appropriate experience, it would be wrong to assign their low performance to the code.

Focusing on the metrics themselves, a special case is the relation between time and correctness. Errors by definition reflect misunderstandings. The question is whether this is due to the difficulty of the code or to misleading beacons. Evidence that time and correctness may actually reflect different concepts is given by Ajami et al. [4]. This study included a comparison of understanding a canonical `for` loop (`for (i=0; i<n; i++)`) with variations in which the initialization, termination condition, or step are varied. The results were that loops counting down took a bit longer, while loops with abnormal initialization or termination caused more errors. The suggested interpretation was that time reflects difficulty, and the error rate reflects a "surprise factor", namely whether the code deviates from expectations. Thus if the code contains misleading elements it may be ill-advised to combine time and correctness scores.

#### *4.2.2 Learning and Fatigue effects*

One confounding effect that deserves special attention is that measurements can change during the experiment. For example, when multiple codes are used the question arises in what order to display them. Using the same order for all experimental subjects reduces variability and enhances comparisons. However, such a consistent order may cause a confounding effect due to learning, fatigue, or dropouts. For example, if unsuccessful subjects feel discouraged and drop out of the experiment, only the more successful subjects will reach the last questions [4]. In other words, a difference in performance on different codes may be the result of their placement in the sequence, rather than a result of the differences we wish to study. The common solution is to randomize the order.

#### *4.2.3 Measurement Technical Issues*

The understanding of short code snippets may take a short time measured in seconds. Thus inaccuracies in the times of beginning and ending the measurement may have an effect. The beginning is typically when the code is first presented, and does not pose a problem. But the ending time may be ambiguous in the sense that it may or may not also include the time to report the answer. Hofmeister et al. explicitly use a two-step system [67]. Their experimental platform first requires subjects to indicate that they have achieved comprehension. It then stops the clock and freezes the code display, and only then opens a window where the subjects can enter their answer. Wilson et al. use an Eclipse plugin which measures the time spent performing different actions to differentiate between time spent comprehending and time spent coding [158].

An alternative to using eye trackers is to employ an experimental platform which displays the code via a "letterbox slit". With this mechanism most of the screen is hidden from view, and only the lines in the slit are visible [70, 67]. The slit may be moved up and down using the arrow keys. This has the significant advantage of enabling online experiments over the Internet, instead of requiring subjects to come physically to the lab where the eye tracking device is set up. However, it is an unnatural setting, which might impair performance. In particular, subjects cannot use peripheral vision to observe the structure of the code and navigate directly to different locations. With the advent of eye tracking software based on webcams[2], this problem will be eliminated.

#### *4.2.4 Premature Theorizing*

The ultimate goal of research on program comprehension is to formalize theories on the cognitive processes which underlie comprehension (e.g. [22,152, 149,131]). These are then expected to inform and facilitate the design of better

[2] For example, GazeRecorder https://gazerecorder.com/.

tools and methodologies for software development. However, not every measurement should lead directly to a cognitive theory. We need to collect a lot of data first. In particular we need multiple replications of existing research, which is the way to increase our confidence in the results, to better define their limitations, and to illuminate their nuances.

## 5 The Experimental Subjects

Previous sections were about technical aspects of program comprehension studies. Despite the various problems and complications that were discussed, these are things that are relatively easy to control. The biggest problem is the human element, namely the experimental subjects. Helpfully, there have been several reviews and guides on this matter (e.g. [78]).

Different people exhibit different levels of performance in all human endeavors, including in code development and program comprehension. Three important high-level factors that affect performance are [27]:

- **Knowledge** — what a developer knows, e.g. the syntax, semantics, and common idioms of a programming language and the background of the application domain;
- **Skill** — the developer's aptitude in applying his or her knowledge in a given situation, and the degree to which this is done automatically or requires effort; and
- **Motivation** — how much the developer actually wants to develop, which affects the effort invested in applying the skill.

As people may have different knowledge, different skills, and different levels of motivation, their performance will differ too. This is exacerbated by the degree of relevance of their knowledge and skills to the experimental task. In the context of academic experiments on program comprehension, using students as subjects has drawn some objections, based on the perception that their skills differ from those of professional developers.

### 5.1 Considerations

Variability among humans is a huge confounding factor, which is hard to assess and control [17,39]. Large individual differences have been reported in various empirical studies, e.g. by Sackman et al., Curtis, and Prechelt [114, 38,107]. This has three main implications. First, it is important to assess the capabilities of experimental subjects and match them to the experiment and tasks as best as is practical. Second, it is important to check whether the variability correlates with demographic variables, and assess whether this affects the external validity of the experiment. Finally, studies have to contend with large variability, and use large enough samples and appropriate statistical methods. In particular, it is desirable to use within-subjects designs over

between-subjects designs, or at least to control for variability by dividing subjects into groups based on perceived differences which might affect the results. Most of the following subsections are elaborations of these considerations.

### *5.1.1 Skill and Experience*

As shown in Fig. 7, programming proficiency and skill can be seen as the general factor which sums up the effects of individual factors like knowledge and experience. Dreyfus and Dreyfus identified the following possible levels [42]:

1. Novice: knows how to apply learned rules to basic situations.
2. Competent: recognizes and uses recurring patterns based on experience.
3. Proficient: prioritizes based on a holistic view of the situation.
4. Expert: experienced enough to do the above intuitively and automatically.

Note that the different levels differ not only in the expected performance but also in the approach taken to solve the programming task. This naturally also affects the interpretation of experimental results. It is therefore important to identify the desired level of proficiency of the experimental subjects, and to screen subjects so that only those with suitable skills participate in the experiment.

However, it is not easy to assess skill [3]. One possible approach is to use a pre-test, namely require subjects to perform a task ahead of the experiment to assess their general skills. However, a single task may not be enough, and there is a danger of interaction between the screening and the experiment. To get a better picture of subjects abilities, Bergersen et al. have suggested a testing regime that can take up to two days [18]. This is not applicable to short experiments, especially if conducted over the Internet. The other extreme is to make do with self assessment of skill, which has the obvious drawback of being subjective [134].

An often used alternative to assessing skill is counting years of experience. The advantage of this approach is that it is less subjective than self assessment and easily applied: one just asks the prospective subjects how many years of programming experience they have. However, such a formulation is ill-defined, as some people count programming during their studies as experience while other do not. It is therefore important to explicitly ask about professional experience, or in other words "real" experience, beyond that obtained during studies.

However, one should note that more experience does not necessarily correlate with higher performance [145,146]. One reason may be that knowledge may be more important for skill than experience [15]. Falessi et al. stress the need to consider not only duration of experience, but whether the experience is relevant and recent [50].

More generally, Ericsson and others stress the importance of what the years of experience were spent on. Performing rote work that does not challenge you and expand your horizons will not improve your skills beyond being able to do

the same thing automatically. To really improve, deliberate practice is needed [46,47]. This means performing challenging work just beyond your comfort zone, receiving feedback that allows you to learn and improve, and doing this over the over again. If this is not done, achievements tend to "flatten out" after several years [95,66,46]. Assuming this is often the case, a useful threshold for tagging subjects as "experienced" can be as low as 3–5 years of professional experience. Requiring more will exclude too many subjects and not provide additional benefits.

In cases where the study requires a distinction between novices and experienced programmers, Feitelson et al. suggest to define three groups [54]:

- Novice students without significant programming experience outside their studies, e.g. in the first or second year of their undergraduate studies and with at most 2 years of programming experience.
- Experienced professionals, with at least 5 years of programming for a living beyond any programming done during their studies.
- All those falling in between the above two groups. These are excluded from the analysis, to sharpen the distinction between students/novices and professionals.

#### *5.1.2 Using Students*

Most studies on software engineering, including those focused on comprehension, loosely target "professional developers". But in practice many studies employ students as subjects, because students are more accessible to academics. Indeed, it is hard to escape the perception that so many studies target "novice" programmers precisely because student subjects are so accessible. The preponderance of such studies raises the question of whether performing experiments with students as subjects is appropriate [50]. Feitelson lists the following potential problems with students [52]:

- By definition students before graduation have not completed their studies. In addition, they may not have fully ingested what they had learned, or hold misconceptions regarding what they have learned [76,85,86]. The implication is that the knowledge at their disposal is not as complete as that of professionals.
- They may not know of commonly used tools or use them ineffectively. This not only affects their performance relative to professionals, but also means that they may use a completely different approach.
- They lack practical experience, which makes it harder for them to find and focus on the heart of the issue. In addition, experience hones skills and facilitates higher performance with less effort.
- Their academic orientation may be misaligned with the needs in industry.

On the other hand students may be more consistent in following instructions, rather than trying to cut to the core in whatever way (including violating the experimental protocol).

It is also important to note that the dichotomy pitting "students" against "professionals" is overly simplistic. Students may have had professional experience in their past or work in parallel with their studies. Graduating students are very close to novice professionals. Consequently, classifying subjects based on their work experience, while far from perfect, is still usually better than classifying them based on their student status.

In particular this means that completely avoiding students as subjects is unwarranted. In fact, when studying how beginners learn to program, students, and even first-year students in particular, are the natural subjects to use. Experiments using students can also be useful to focus the research and to debug experimental procedures [151,9]. And in many cases experiments with students yield the same relative results as experiments with professionals, in the sense that the relations between the different treatments are the same. Only the absolute results are different, with professionals usually performing better (e.g. [90,11]).

Note too that the common career path for graduating students is to seek industrial positions. So students close to graduation are essentially the same as beginning professionals. However, many professionals do not have an academic background: for example, in the 2021 StackOverflow developer survey[3], nearly 60% said they learned to code from online blogs and videos, 40% cited online courses or certification, and less than 54% said they learned at a school (the sum is larger than 100% as they noted all that apply). So students represent only about half of developers.

One situation in which students indeed should not be used is as proxies for experts. Too many studies aim to expose differences between novices and experts, and use beginning undergraduates as the novices and graduate students or even third-year students as the experts. This is wrong. They are slightly more advanced students, but not programming experts.

### *5.1.3 Ensuring Motivation*

As noted above, motivation is required for the experimental subjects to apply their skill in performing the tasks of the experiment. Results obtained from unmotivated subjects may distort the data. Motivation is also important for the recruitment of subjects in the first place.

Various steps can be taken to increase the motivation of subjects to participate in an experiment. Major incentives include the following:

- In many studies the only incentive is interest in the study and its results. for professionals, this can be based on having confronted situations similar to those in the experiment in day-to-day work. Experiments can also provide a welcome and thought-provoking diversion from routine work. In both cases, the interest in the experiment should not be tempered by hardships such as excessive length. Keeping experiments as short as possible — as short as 10 to 15 minutes — helps recruit and retain subjects. The price

[3] https://insights.stackoverflow.com/survey/2021

is that the length of code or number of treatments used in the experiment may need to be reduced.
- A common approach to incentivizing subjects is to pay them, either paying a small sum directly or by holding a raffle for a larger sum. Naturally the payment should not be so large as to incentivize people who are not suitable to pose as subjects. Payments like this are often useful with students, but probably less so with well-paid professionals, unless they are actually hired as part of a large-scale experiment.
- In some cases the experimental subjects do the experiment authors a favor by participating. For example it is common practice to ask friends and colleagues to take part in pilot studies used to validate and adjust experimental materials and procedures. Inviting colleagues and acquaintances to participate in the final experiment, and snowballing from there, is also not uncommon. This is generally a useful way to recruit participants. However it might have the disadvantage of narrowing the field to subjects with similar characteristics.

The most motivated potential subjects are the experiment authors themselves. However, self experiments should be avoided, as the authors not only may have a conflict of interest, but their performance may also be influenced by their knowledge of the experiment design.

#### *5.1.4 Effect of Demographics*

A recurring theme when considering experimental subjects is whether demographic variables, mainly sex and age, may explain some of the variability. Some studies have reported observed differences between men and women [81, 125, 48]. Others have found no such differences [4, 54]. At present it seems that the differences, when and if they exist, are not major, but this deserves further study.

#### *5.1.5 Ethics in Research*

An experiment on how people understand code is an experiment with human subjects, and as such must follow research ethics guidelines. The basic principles for ethical research were laid out in the Belmont report in 1979 [150]. While this was done in the context of bio-medial research, two main ideas carry over to software engineering experiments:

- Subjects should be respected, implying that experimental subjects should be informed about the experiment and are entitled to decide for themselves whether to participate in it. For example, posing as a ranking service to collect data on developers' aptitude is unethical. Likewise, students should not be coerced to participate in an experiment by their professors.
- In performing the experiments the researchers are obliged to do no harm, and should avoid the danger of jeopardizing subjects' well-being in any way. For example, identifying subjects who made stupid mistakes is unethical.

Ethical compliance is usually ensured by the practices of obtaining informed consent from experimental subjects, allowing them to leave the experiment at any time, and not collecting any identifying information.

Carver et al. point out that special considerations apply when students are used as subjects. One needs to remember that the students are there for an education, and participating in an experiment can affect this education [29]. It is up to the researchers to ensure that this effect is for the good. More generally, care should be taken to ensure non-coercing participation and to limit stress.

## 5.2 Pitfalls

### *5.2.1 Subjects Unsuited for the Study*

A potentially significant problem may be the failure to exclude subjects who are unsuited for the task done in the experiment. This should be verified as part of the initial demographic screening. However, it is not easy to think in advance of all the factors that need to be checked.

An obvious exclusion criterion is that subjects should be well-versed in the programming language used. They should be excluded if they lack knowledge needed to perform well in the study, e.g. knowledge about certain technologies. However, Carver et al. suggest that this last deficiency can be corrected quickly by first observing someone else perform the experiment while using the required technology [28].

Inexperienced subjects should not be used when the research involves not just basic or technical knowledge but performance honed by practice. And practice can have an effect on many different things. For example, naming or documentation practices may change after one has first-hand experience suffering from the practices of others. Indeed, names given by experienced developers have been found to differ from those given by inexperienced students [54].

There can also be difficulties unrelated to knowledge. For example, contact lenses and downward pointing eyelashes appear to reduce accuracy in eye tracking studies [96]. Eye glasses, on the other hand, are fine.

Last, subjects repeating the experiment should most probably also be excluded. But if no identifying information is collected, this has to rely on subject self reporting. To enable such reporting, a question needs to be included in the demographic screening.

As an example, Hofmeister et al. report having used the following exclusions criteria to exclude 63 of 135 participants in a study, leaving only 72 valid ones [67]:

- Self rating of language proficiency as being 1–3 on a scale of 1–6, for both German (the language of the instructions) and English (the language of code variable names and comments).

- Self rating of C# skills of 1–3 on a scale of 1–5, or less than 1 year of practical use of the language.
- Admitting having been distracted or not having worked conscientiously.
- Too low performance, e.g. taking more than 10 minutes to perform a trial.
- Having already participated in the study or in a pilot study.

### *5.2.2 Lack of Relevant Knowledge*

A special case of unsuitable subjects that deserves further attention is when subjects lack relevant knowledge. Developers often have different levels of knowledge in different pertinent dimensions of knowledge. The most important distinction is between the technical dimension and the domain dimension [121]. Both should be checked to ascertain that the study participants indeed have the required background.

The technical dimension involves knowledge about the programming language, the development environment, the process workflows, etc. A minimal requirement in experiments is that subjects be proficient in the language in which the code is written. Note that proficiency is more than mere working knowledge of the language, and includes being acquainted with the programming culture and ecosystem around the language. This should therefore be included in the screening of subjects.

Domain knowledge is about the background of the code or application. This can mean general knowledge about the application itself: what exactly it is supposed to do, why, and how it fits into the bigger picture. But in many cases the more important background concerns the whole domain. For example, understanding a scientific code which performs physics calculations would typically require a knowledge of the underlying physics, and understanding a banking investments application would require deep knowledge of the financial system. This is the reason that studies are often conducted using general code (such as utility libraries) and not specialized code.

Note that these two dimensions are required in different amounts for different tasks. For example, technical knowledge is sufficient for fixing a technical bug like a null pointer reference, but domain knowledge is crucial for providing suitable context in a code summarization task. In addition, it is important to note that knowledge dimensions may interact with tasks. According to von Mayrhauser and Vans, adaptive maintenance tasks require much more domain knowledge than program knowledge [154]. Corrective maintenance and the development of new features, on the other hand, require much more program knowledge than domain knowledge.

### *5.2.3 Differences in Definition of Levels*

Many studies attempt to perform a comparison between novices and experts. However, developers often have different levels of knowledge in different pertinent dimensions of knowledge. Therefore any uni-dimensional classification into “novices” and “experts” is compromised. In addition, in some cases the

**Table 1** *Examples of definitions of novice/intermediate/expert subjects.*

| Ref. | Level 1 | Level2 | Level 3 |
|---|---|---|---|
| [2] | low: grades below threshold | high: grades above threshold | |
| [11] | novice: <24 months programming experience | intermediate: >24 months programming experience | |
| [20] | novice: 2nd year students | expert: industry professionals considered experts | |
| [26] | novice: inexperienced student | experienced professional | |
| [31] | bachelor student | master student | PhD student |
| [37] | novice: 2nd term CS students | intermediate: junior/senior CS students | advanced: graduate CS students + faculty |
| [89] | beginner: starting 1st course | intermediate: finishing 1st course | expert: teaching course, with 400 hr experience |
| [115] | 3rd year bachelor student | professional | |
| [119] | novice: 1–6 months job experience | experienced rated low by supervisors | experienced rated high by supervisors |
| [144] | novice: end of 1st programming course | advanced: completed 3 programming courses | |
| [156] | novice: undergrad CS majors | expert: 2nd year graduate students | manager: from industry |

differences between the levels may not be significant enough to make a difference, for example when comparing 3rd year students with masters students.

Another problem is that the definitions used by different researchers differ considerably, making any comparison between different studies practically meaningless (see Table 1). For example, many use graduate students, or even students towards the end of their first degree, as "experienced". This might be true relative to freshmen in their first year, but does not reflect experience gained in a few years on the job. A possible approach is to classify students by year during undergraduate studies or as "advanced" for graduates, whereas professionals would be classified into "novices" (say up to a year of on-the-job experience) and "experienced" (3 or more years). Note that this leaves a gap between the groups as suggested above.

### *5.2.4 Unmotivated Subjects*

The performance on any work task may be affected by motivation, interest, and mood: happy developers are more productive and create better-quality code [62,63,61]. Personality also has an effect [64]. This naturally also applies in experiments. But if the experiment is conducted on-line, you have no way to know how the experimental subjects are feeling, or whether they were distracted. And motivation when participating in experiments may not be the same as in real work. Hofmeister et al. therefore suggest to perform a final debriefing at the end of the experiment and exclude subjects who report that they were not working conscientiously or were distracted [67].

## 6 Conclusions

Methodological discussions on software engineering experiments have typically focused mainly on experimental design, statistical tests, and reporting guidelines (e.g. [74,73,159,130]). This focus mirrors the reaction to the reproducibility crisis in psychological research [143]. But validity and reproducibility are compromised not only by flaws in the statistics and the reporting. The conclusion of an otherwise solid study can also be jeopardized by inadvertent nuances in the experimental materials and the experimental procedure.

In the context of experiments on program comprehension, very little discussion has appeared in the literature on what exactly we mean when we say "the subject understands the code", and how the code and tasks we use affect this issue. We need more work on such methodological issues, and better reporting not only of the details but also of the considerations involved in selecting the code and the tasks. Table 2 summarizes the main points made in the previous sections. Many of these points may seem obvious. But the literature is rife with examples of good research papers that did not take some of these considerations into account or failed on some pitfall.

The purpose of this paper is not to dictate the "right" way to do research. Its purpose is to raise awareness to the myriad considerations that are involved in experiments on program comprehension, and especially to the side effects that methodological decisions may have. Such awareness is needed mainly to increase the volume of discussion of methodological issues, including methodological differences. Awareness of differences is important for better understanding of how the results of different studies can be compared to each other, and how they complement each other. This, together with multiple divergent replications of previous work, is the path to a deeper understanding of how code is understood.

## References

1. M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol, "*An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension*". In

**Table 2** Tentative checklist for program comprehension experiments.

| | Related to Code |
|---|---|
| ☐ | Use minimal scope to avoid diluting the experiment; but if needed, "minimal" may be a full system |
| ☐ | Do not pad code to achieve a pre-specified length |
| ☐ | Do not use trivial code, contrived code, or overly sophisticated code |
| ☐ | Use real code unless need to write code specifically for the experiment; make sure code compiles and runs correctly |
| ☐ | Beware of misleading code and in particular misleading names |
| ☐ | Avoid well-known recognizable code |
| ☐ | Use `a`, `b`, `c`, etc. or general terms like `num` to obfuscate names |
| ☐ | Apply judgment rather than mechanical solutions for specific issues such as the use of abbreviations |
| ☐ | Do not include dead code |
| ☐ | Use consistent style based on IDE defaults |
| ☐ | Ensure that the code is appropriate for the task but does not give it away |
| | Related to Task |
| ☐ | Use a recognition task for research on identifying tokens and structure |
| ☐ | Use a parsing task for research on understanding syntax |
| ☐ | Use an interpretation task (what does this print) for research on semantics of individual instructions |
| ☐ | Use a comprehension task (name this function) for research on global semantics |
| ☐ | Use a semantic bug-fixing task for research on understanding the mechanics of the code |
| ☐ | Use a localization or modification task for research on understanding large scale structure |
| ☐ | Use a black-box task for research on understanding APIs |
| ☐ | Use a refactoring task for research on understanding design |
| ☐ | Beware of tasks that can be circumvented |
| ☐ | Use a well-defined base task as control |
| ☐ | Provide a suitable and convenient working environment |
| | Related to Measurement |
| ☐ | Plan how to accurately measure time |
| ☐ | Plan how to judge the correctness of answers |
| ☐ | Decide what to do if a wrong answer is given |
| ☐ | Consider whether and how to combine time measurements with correctness assessments |
| ☐ | Consider the use of eye tracking or other biophysical measurements |
| ☐ | Beware of non-representative results at beginning of experiment |
| ☐ | Beware of fatigue, learning, and dropout effects |
| | Related to Subjects |
| ☐ | Make sure subjects have the appropriate levels of knowledge and skill for the experiment |
| ☐ | Consider whether and how the subjects can be divided into "novices" and "experienced" |
| ☐ | Do not over-obsess about subjects being students |
| ☐ | Use preliminary tasks to test skill and knowledge |
| ☐ | Try to ensure that subjects are motivated |
| ☐ | Consider the effect of demographic factors |
| ☐ | Exclude subjects who are not suited to the task |
| ☐ | Follow ethical guidelines |

15th *European Conf. Softw. Maintenance & Reengineering*, pp. 181–190, Mar 2011, DOI: 10.1109/CSMR.2011.24.
2. 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.
3. W. K. Adams and C. E. Wieman, "*Development and validation of instruments to measure learning of expert-like thinking*". *Intl. J. Science Education* **33(9)**, pp. 1289–1312, Jun 2011, DOI: 10.1080/09500693.2010.512369.
4. S. Ajami, Y. Woodbridge, and D. G. Feitelson, "*Syntax, predicates, idioms — what really affects code complexity?*" *Empirical Softw. Eng.* **24(1)**, pp. 287–328, Feb 2019, DOI: 10.1007/s10664-018-9628-3.
5. V. Arnaoudova, M. Di Penta, and G. Antoniol, "*Linguistic antipatterns: What they are and how developers perceive them*". *Empirical Softw. Eng.* **21(1)**, pp. 104–158, Feb 2016, DOI: 10.1007/s10664-014-9350-8.
6. 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.
7. V. R. Basili and R. W. Selby, "*Comparing the effectiveness of software testing strategies*". *IEEE Trans. Softw. Eng.* **SE-13(12)**, pp. 1278–1296, Dec 1987, DOI: 10.1109/TSE.1987.232881.
8. V. R. Basili, R. W. Selby, and D. H. Hutchens, "*Experimentation in software engineering*". *IEEE Trans. Softw. Eng.* **SE-12(7)**, pp. 733–743, Jul 1986, DOI: 10.1109/TSE.1986.6312975.
9. V. R. Basili and M. V. Zelkowitz, "*Empirical studies to build a science of computer science*". *Comm. ACM* **50(11)**, pp. 33–37, Nov 2007, DOI: 10.1145/1297797.1297819.
10. J. Bauer, J. Siegmund, N. Peitek, J. C. Hofmeister, and S. Apel, "*Indentation: Simply a matter of style or support for program comprehension?*" In 27th *Intl. Conf. Program Comprehension*, pp. 154–164, May 2019, DOI: 10.1109/ICPC.2019.00033.
11. R. Bednarik, N. Myller, E. Sutinen, and M. Tukiainen, "*Effects of experience on gaze behavior during program animation*". In 17th *Workshop of Psychology of Programming Interest Group*, pp. 49–61, Jun 2005.
12. R. Bednarik and M. Tukiainen, "*An eye-tracking methodology for characterizing porgram comprehension processes*". In 4th *Symp. Eye Tracking Res. & App.*, pp. 125–132, Mar 2006, DOI: 10.1145/1117309.1117356.
13. R. Bednarik et al., "*EMIP: The eye movements in programming dataset*". *Sci. Comput. Programming* **198**, art. 102520, Oct 2020, DOI: 10.1016/j.scico.2020.102520.
14. G. Beniamini, S. Gingichashvili, A. Klein Orbach, and D. G. Feitelson, "*Meaningful identifier names: The case of single-letter variables*". In 25th *Intl. Conf. Program Comprehension*, pp. 45–54, May 2017, DOI: 10.1109/ICPC.2017.18.
15. 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.
16. G. R. Bergersen, J. E. Hannay, D. I. K. Sjøberg, T. Dybå, and A. Karahasanović, "*Inferring skill from tests of programming performance: Combining time and quality*". In 5th *Intl. Symp. Empirical Softw. Eng. & Measurement*, pp. 305–314, Sep 2011, DOI: 10.1109/ESEM.2011.39.
17. G. R. Bergersen and D. I. K. Sjøberg, "*Evaluating methods and technologies in software engineering with respect to developer's skill level*". In 16th *Intl. Conf. Evaluation & Assessment in Softw. Eng.*, pp. 101–110, May 2012, DOI: 10.1049/ic.2012.0013.
18. 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.
19. T. J. Biggerstaff, "*Design recovery for maintenance and reuse*". *Computer* **22(7)**, pp. 36–49, Jul 1989, DOI: 10.1109/2.30731.
20. 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.

21. 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.
22. 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.
23. R. E. Brooks, "*Studying programmer behavior experimentally: The problems of proper methodology*". *Comm. ACM* **23(4)**, pp. 207–213, Apr 1980, DOI: 10.1145/358841.358847.
24. 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.
25. R. P. L. Buse and W. R. Weimer, "*Learning a metric for code readability*". *IEEE Trans. Softw. Eng.* **36(4)**, pp. 546–558, Jul/Aug 2010, DOI: 10.1109/TSE.2009.70.
26. T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "*Eye movements in code reading: Relaxing the linear order*". In 23rd *Intl. Conf. Program Comprehension*, pp. 255–265, May 2015, DOI: 10.1109/ICPC.2015.36.
27. J. P. Campbell, R. A. McCloy, S. H. Oppler, and C. E. Sager, "*A theory of performance*". In *Personnel Selection in Organizations*, N. Schmitt, W. C. Borman, and Associates (eds.), pp. 35–70, Jossey-Bass Pub., 1993.
28. J. Carver, F. Shull, and V. Basili, "*Observational studies to accelerate process experience in classroom studies: An evaluation*". In *Intl. Symp. Empirical Softw. Eng.*, pp. 72–79, Sep 2003, DOI: 10.1109/ISESE.2003.1237966.
29. J. C. Carver, L. Jaccheri, S. Morasca, and F. Shull, "*A checklist for integrating student empirical studies with research and teaching goals*". *Empirical Softw. Eng.* **15(1)**, pp. 35–59, Feb 2010, DOI: 10.1007/s10664-009-9109-9.
30. R. Cates, N. Yunik, and D. G. Feitelson, "*Does code structure affect comprehension? on using and naming intermediate variables*". In 29th *Intl. Conf. Program Comprehension*, pp. 118–126, May 2021, DOI: 10.1109/ICPC52881.2021.00020.
31. M. Ceccato, M. Di Penta, P. Falcarin, F. Ricca, M. Torchiano, and P. Tonella, "*A family of experiments to assess the effectiveness and efficiency of source code obfuscation techniques*". *Empirical Softw. Eng.* **19(4)**, pp. 1040–1074, Aug 2014, DOI: 10.1007/s10664-013-9248-x.
32. M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "*Let's go to the whiteboard: How and why software developers use drawings*". In *SIGCHI Conf. Human Factors in Comput. Syst.*, pp. 557–566, Apr 2007, DOI: 10.1145/1240624.1240714.
33. E. J. Chikofsky and J. H. Cross II, "*Reverse engineering and design recovery: A taxonomy*". *IEEE Softw.* **7(1)**, pp. 13–17, Jan 1990, DOI: 10.1109/52.43044.
34. C. Cook, W. Bregar, and D. Foote, "*A preliminary investigation of the use of the cloze procedure as a measure of program understanding*". *Inf. Process. & Management* **20(1–2)**, pp. 199–208, 1984, DOI: 10.1016/0306-4573(84)90050-5.
35. B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "*A systematic survey of program comprehension through dynamic analysis*". *IEEE Trans. Softw. Eng.* **35(5)**, pp. 684–702, Sep/Oct 2009, DOI: 10.1109/TSE.2009.28.
36. R. Couceiro, G. Duarte, J. Durães, J. Castelhano, C. Duarte, C. Teixeira, M. Castelo Branco, P. de Carvalho, and H. Madeira, "*Biofeedback augmented software engineering: Monitoring of programmers' mental effort*". In 41st *Intl. Conf. Softw. Eng.*, pp. 37–40, May 2019, DOI: 10.1109/ICSE-NIER.2019.00018. (NIER track).
37. M. E. Crosby, J. Scholtz, and S. Wiedenbeck, "*The roles beacons play in comprehension for novice and expert programmers*". In 14th *Workshop Psychology of Programming Interest Group*, pp. 58–73, Jun 2002.
38. B. Curtis, "*Substantiating programmer variability*". *Proc. IEEE* **69(7)**, p. 846, Jul 1981, DOI: 10.1109/PROC.1981.12088.
39. B. Curtis, "*A career spent wading through industry's empirical ooze*". In 2nd *Intl. Workshop Conducting Empirical Studies in Industry*, pp. 1–2, Jun 2014, DOI: 10.1145/2593690.2593699.
40. 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.
41. E. W. Dijkstra, "*Go To statement considered harmful*". *Comm. ACM* **11(3)**, pp. 147–148, Mar 1968, DOI: 10.1145/362929.362947.

42. S. E. Dreyfus and H. L. Dreyfus, *A Five-Stage Model of the Mental Activities Involved in Directed Skill Acquisition*. Tech. Rep. ORC-80-2, Operations Research Center, University of California, Berkeley, Feb 1980.
43. W. H. DuBay, "*The principles of readability*". URL http://www.impact-information.com/impactinfo/readability02.pdf, Aug 2004.
44. A. Dunsmore and M. Roper, *A Comparative Evaluation of Program Comprehension Measures*. Tech. Rep. EFoCS-35-2000, University of Strathclyde, 2000.
45. A. Dunsmore, M. Roper, and M. Wood, "*The role of comprehension in software inspection*". *J. Syst. & Softw.* **52(2–3)**, pp. 121–129, Jun 2000, DOI: 10.1016/S0164-1212(99)00138-7.
46. K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer, "*The role of deliberate practice in the acquisition of expert performance*". *Psychological Rev.* **100(3)**, pp. 363–406, Jul 1993, DOI: 10.1037/0033-295X.100.3.363.
47. K. A. Ericsson, M. J. Prietula, and E. T. Cokely, "*The making of an expert*". *Harvard Business Rev.* Jul-Aug 2007.
48. A. Etgar, R. Friedman, S. Haiman, D. Perez, and D. G. Feitelson, "*The effect of information content and length on name recollection*". In 30th *Intl. Conf. Program Comprehension*, May 2022.
49. S. Fakhoury, D. Roy, Y. Ma, V. Arnaoudova, and O. Adesope, "*Measuring the impact of lexical and structural inconsistencies on developers' cognitive load during bug localization*". *Empirical Softw. Eng.* **25(3)**, pp. 2140–2178, May 2020, DOI: 10.1007/s10664-019-09751-4.
50. D. Falessi, N. Juristo, C. Wohlin, B. Turhan, J. Münch, A. Jedlitschka, and M. Oivo, "*Empirical software engineering experts on the use of students and professionals in experiments*". *Empirical Softw. Eng.* **23(1)**, pp. 452–489, Feb 2018, DOI: 10.1007/s10664-017-9523-3.
51. 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.
52. 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].
53. D. G. Feitelson, "*Considerations and pitfalls in controlled experiments on code comprehension*". In 29th *Intl. Conf. Program Comprehension*, pp. 106–117, May 2021, DOI: 10.1109/ICPC52881.2021.00019.
54. D. G. Feitelson, A. Mizrahi, N. Noy, A. Ben Shabat, O. Eliyahu, and R. Sheffer, "*How developers choose names*". *IEEE Trans. Softw. Eng.* **48(1)**, pp. 37–52, Jan 2022, DOI: 10.1109/TSE.2020.2976920.
55. B. Floyd, T. Santander, and W. Weimer, "*Decoding the representation of code in the brain: An fMRI study of code review and expertise*". In 39th *Intl. Conf. Softw. Eng.*, pp. 175–186, May 2017, DOI: 10.1109/ICSE.2017.24.
56. M. Fowler, *Refactoring: Improving the Design of Existing Code*. Pearson Education, Inc., 2nd ed., 2019.
57. T. Fritz, A. Begel, S. C. Müller, S. Yigit-Elliott, and M. Züger, "*Using psycho-physiological measures to assess task difficulty in software development*". In 36th *Intl. Conf. Softw. Eng.*, pp. 402–413, May 2014, DOI: 10.1145/2568225.2568266.
58. Y. Geffen and S. Maoz, "*On method ordering*". In 24th *Intl. Conf. Program Comprehension*, May 2016, DOI: 10.1109/ICPC.2016.7503711.
59. 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.
60. D. Gopstein, J. Iannacone, Y. Yan, L. DeLong, Y. Zhuang, M. K.-C. Yeh, and J. Cappos, "*Understanding misunderstanding in source code*". In 11th *ESEC/FSE*, pp. 129–139, Aug 2017, DOI: 10.1145/3106237.3106264.
61. D. Graziotin, F. Fagerholm, X. Wang, and P. Abrahamsson, "*What happens when software developers are (un)happy*". *J. Syst. & Softw.* **140**, pp. 32–47, Jun 2018, DOI: 10.1016/j.jss.2018.02.041.
62. D. Graziotin, X. Wang, and P. Abrahamsson, "*Software developers, moods, emotions, and performance*". *IEEE Softw.* **31(4)**, pp. 24–27, Jul/Aug 2014, DOI: 10.1109/MS.2014.94.

63. D. Graziotin, X. Wang, and P. Abrahamsson, "*How do you feel, developer? an explanatory theory of the impact of affects on programming performance*". *peerJ Comput. Sci.* **1**, art. e18, Aug 2015, DOI: 10.7717/peerj-cs.18.
64. J. E. Hannay, "*Personality, intelligence, and expertise: Impacts on software development*". In *Making Software*, A. Oram and G. Wilson (eds.), pp. 79–110, O'Reilly Media Inc., 2011.
65. C. Hannebauer, M. Hesenius, and V. Gruhn, "*Does syntax highlighting help programming novices?*" *Empirical Softw. Eng.* **23(5)**, pp. 2795–2828, Oct 2018, DOI: 10.1007/s10664-017-9579-0.
66. 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.
67. J. C. Hofmeister, J. Siegmund, and D. V. Holt, "*Shorter identifier names take longer to comprehend*". *Empirical Softw. Eng.* **24(1)**, pp. 417–443, Feb 2019, DOI: 10.1007/s10664-018-9621-x.
68. N. Hollmann and S. Hanenberg, "*An empirical study on the readability of regular expressions: Textual versus graphical*". In *Working Conf. Softw. Visualization*, pp. 74–84, Sep 2017, DOI: 10.1109/VISSOFT.2017.27.
69. A. A. Ivanova, S. Srikant, Y. Sueoka, H. H. Kean, R. Dhamala, U.-M. O'Reilly, M. U. Bers, and E. Fedorenko, "*Comprehension of computer code relies primarily on domain-general executive brain regions*". *eLife* **9**, art. e58906, Dec 2020, DOI: 10.7554/eLife.58906.
70. A. R. Jansen, A. F. Blackwell, and K. Marriott, "*A tool for tracking visual attention: The restricted focus viewer*". *Behavior Research Methods, Instruments, & Comput.* **35(1)**, pp. 57–69, Feb 2003, DOI: 10.3758/BF03195497.
71. 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.
72. 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.
73. A. Jedlitschka and D. Pfahl, "*Reporting guidelines for controlled experiments in software engineering*". In *Intl. Symp. Empirical Softw. Eng.*, pp. 95–104, Nov 2005, DOI: 10.1109/ISESE.2005.1541818.
74. N. Juristo and A. M. Moreno, *Basics of Software Engineering Experimentation*. Kluwer, 2001.
75. N. Juristo, S. Vegas, M. Solari, S. Abrahao, and I. Ramos, "*Comparing the effectiveness of equivalence partitioning, branch testing and code reading by stepwise abstraction applied by subjects*". In 5th *Intl. Conf. Software Testing, Verification, & Validation*, pp. 330–339, Apr 2012, DOI: 10.1109/ICST.2012.113.
76. L. C. Kaczmarczyk, E. R. Petrick, J. P. East, and G. L. Herman, "*Identifying student misconceptions of programming*". In 41st *SIGCSE Tech. Symp. Comput. Sci. Ed.*, pp. 107–111, Mar 2010, DOI: 10.1145/1734263.1734299.
77. D. Kahneman, *Attention and Effort*. Prantice-Hall, 1973.
78. A. J. Ko, T. D. LaToza, and M. M. Burnett, "*A practical guide to controlled experiments of software engineering tools with human participants*". *Empirical Softw. Eng.* **20(1)**, pp. 110–141, Feb 2015, DOI: 10.1007/s10664-013-9279-3.
79. P. Kruchten, "*The 4+1 view model of architecture*". *IEEE Softw.* **12(6)**, pp. 42–50, Nov 1995, DOI: 10.1109/52.469759.
80. R. Krueger, Y. Huang, X. Liu, T. Santander, W. Weimer, and K. Leach, "*Neurological divide: An fMRI study of prose and code writing*". In 42nd *Intl. Conf. Softw. Eng.*, pp. 678–690, Oct 2020, DOI: 10.1145/3377811.3380348.
81. D. Lawrie, C. Morrell, H. Field, and D. Binkley, "*What's in a name? a study of identifiers*". In 14th *Intl. Conf. Program Comprehension*, pp. 3–12, Jun 2006, DOI: 10.1109/ICPC.2006.51.
82. O. Levy and D. G. Feitelson, "*Understanding large-scale software systems — structure and flows*". *Empirical Softw. Eng.* **26(3)**, art. 48, May 2021, DOI: 10.1007/s10664-021-09938-8.

83. B. P. Lientz, E. B. Swanson, and G. E. Tompkins, "*Characteristics of application software maintenance*". *Comm. ACM* **21(6)**, pp. 466–471, Jun 1978, DOI: 10.1145/359511.359522.
84. D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, "*Mental models and software maintenance*". *J. Syst. & Softw.* **7(4)**, pp. 341–355, Dec 1987, DOI: 10.1016/0164-1212(87)90033-1.
85. L. Ma, J. Ferguson, M. Roper, and M. Wood, "*Investigating the viability of mental models held by novice programmers*". In 38th *SIGCSE Symp. Comput. Sci. Education*, pp. 499–503, Mar 2007, DOI: 10.1145/1227504.1227481.
86. S. Madison and J. Gifford, "*Modular programming: Novice misconceptions*". *J. Res. Tech. Ed.* **34(3)**, pp. 217–229, 2002, DOI: 10.1080/15391523.2002.10782346.
87. R. C. Martin, *Clean Code: A Handbook of Agile Software Craftmanship*. Prentice Hall, 2009.
88. T. McCabe, "*A complexity measure*". *IEEE Trans. Softw. Eng.* **SE-2(4)**, pp. 308–320, Dec 1976, DOI: 10.1109/TSE.1976.233837.
89. K. B. McKeithen, J. S. Reitman, H. H. Reuter, and S. C. Hirtle, "*Knowledge organization and skill differences in computer programmers*". *Cognitive Psychol.* **13(3)**, pp. 307–325, Jul 1981, DOI: 10.1016/0010-0285(81)90012-8.
90. D. A. McMeekin, B. R. von Konsky, M. Robey, and D. J. A. Cooper, "*The significance of participant experience when evaluating software inspection techniques*". In *Australian Softw. Eng. Conf.*, pp. 200–209, Apr 2009, DOI: 10.1109/ASWEC.2009.13.
91. B. Meyer, "*Applying "design by contract"*". *Computer* **25(10)**, pp. 40–51, Oct 1992, DOI: 10.1109/2.161279.
92. R. J. Miara, J. A. Musselman, J. A. Navarro, and B. Shneiderman, "*Program indentation and comprehensibility*". *Comm. ACM* **26(11)**, pp. 851–867, Nov 1983, DOI: 10.1145/182.358437.
93. M. Nagappan, R. Robbes, Y. Kamei, E. Tanter, S. McIntosh, A. Mockus, and A. E. Hassan, "*An empirical study of goto in C code from GitHub repositories*". In 10th *ESEC/FSE*, pp. 404–414, Aug 2015, DOI: 10.1145/2786805.2786834.
94. N. Nagappan, T. Ball, and A. Zeller, "*Mining metrics to predict component failures*". In 28th *Intl. Conf. Softw. Eng.*, pp. 452–461, May 2006, DOI: 10.1145/1134285.1134349.
95. 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.
96. M. Nyström, R. Andersson, K. Holmqvist, and J. van der Weijer, "*The influence of calibration method and eye physiology on eyetracking data quality*". *Behavioral Res. Meth.* **45(1)**, pp. 272–288, Mar 2013, DOI: 10.3758/s13428-012-0247-4.
97. U. Obaidellah, M. Al Haek, and P. C.-H. Cheng, "*A survey on the usage of eye-tracking in computer programming*". *ACM Comput. Surv.* **51(1)**, art. 5, Jan 2018, DOI: 10.1145/3145904.
98. D. Oliveira, R. Bruno, F. Madeiral, and F. Castor, "*Evaluating code readability and legibility: An examination of human-centric studies*". In *Intl. Conf. Softw. Maintenance & Evolution*, pp. 348–359, Oct 2020, DOI: 10.1109/ICSME46990.2020.00041.
99. P. W. Oman and C. R. Cook, "*Typographic style is more than cosmetic*". *Comm. ACM* **33(5)**, pp. 506–520, May 1990, DOI: 10.1145/78607.78611.
100. A. Orso, S. Sinha, and M. J. Harrold, "*Effects of pointers on data dependences*". In 9th *IEEE Intl. Workshop Program Comprehension*, pp. 39–49, May 2001, DOI: 10.1109/WPC.2001.921712.
101. D. L. Parnas, "*On the criteria to be used in decomposing systems into modules*". *Comm. ACM* **15(12)**, pp. 1053–1058, Dec 1972, DOI: 10.1145/361598.361623.
102. D. L. Parnas, P. C. Clements, and D. M. Weiss, "*The modular structure of complex systems*". *IEEE Trans. Softw. Eng.* **SE-11(3)**, pp. 259–266, Mar 1985, DOI: 10.1109/TSE.1985.232209.
103. J. W. Paulson, G. Succi, and A. Eberlein, "*An empirical study of open-source and closed-source software products*". *IEEE Trans. Softw. Eng.* **30(4)**, pp. 246–256, Apr 2004, DOI: 10.1109/TSE.2004.1274044.
104. S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "*Evaluating and improving fault localization*". In 39th *Intl. Conf. Softw. Eng.*, pp. 609–620, May 2017, DOI: 10.1109/ICSE.2017.62.

105. N. Pennington, "*Stimulus structures and mental representations in expert comprehension of computer programs*". *Cognitive Psychology* **19(3)**, pp. 295–341, Jul 1987, DOI: 10.1016/0010-0285(87)90007-7.
106. C. Politowski, F. Khomh, S. Romano, G. Scanniello, F. Petrillo, Y.-G. Guéhéneuc, and A. Maiga, "*A large scale empirical study of the impact of Spaghetti Code and Blob anti-patterns on program comprehension*". *Inf. & Softw. Tech.* **122**, art. 106278, June 2020, DOI: 10.1016/j.infsof.2020.106278.
107. 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.
108. H. C. Purchase, L. Colpoys, M. McGill, and D. Carrington, "*UML collaboration diagram syntax: An empirical study of comprehension*". In 1st *Intl. Workshop Visualizing Softw. for Understanding & Analysis*, pp. 13–22, Jun 2002, DOI: 10.1109/VISSOF.2002.1019790.
109. S. Raghunathan, A. Prasad, B. K. Mishra, and H. Chang, "*Open source versus closed source: Software quality in monopoly and competitive markets*". *IEEE Trans. Syst., Man, & Cybernetics* **35(6)**, pp. 903–918, Nov 2005, DOI: 10.1109/TSMCA.2005.853493.
110. 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.
111. V. Rajlich and N. Wilde, "*The role of concepts in program comprehension*". In 10th *IEEE Intl. Workshop Program Comprehension*, pp. 271–278, Jun 2002, DOI: 10.1109/WPC.2002.1021348.
112. E. S. Raymond, "*The cathedral and the bazaar*". URL www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar, 2000.
113. T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "*How do professional developers comprehend software?*" In 34th *Intl. Conf. Softw. Eng.*, pp. 255–265, Jun 2012, DOI: 10.1109/ICSE.2012.6227188.
114. 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.
115. F. Salviulo and G. Scanniello, "*Dealing with identifiers and comments in source code comprehension and maintenance: Results from an ethnographically-informed study with students and professionals*". In 18th *Intl. Conf. Evaluation & Assessment in Softw. Eng.*, art. 48, May 2014, DOI: 10.1145/2601248.2601251.
116. S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vśquez, D. Poshyvanyk, and R. Oliveto, "*Automatically assessing code understandability*". *IEEE Trans. Softw. Eng.* **47(3)**, pp. 595–613, Mar 2021, DOI: 10.1109/TSE.2019.2901468. (early access).
117. S. Scalabrino, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "*Improving code readability models with textual features*". In 24th *Intl. Conf. Program Comprehension*, May 2016, DOI: 10.1109/ICPC.2016.7503707.
118. A. Schankin, A. Berger, D. V. Holt, J. C. Hofmeister, T. Riedel, and M. Beigl, "*Descriptive compound identifier names improve source code comprehension*". In 26th *Intl. Conf. Program Comprehension*, pp. 31–40, May 2018, DOI: 10.1145/3196321.3196332.
119. K. D. Schenk, N. P. Vitalari, and K. S. Davis, "*Differences between novice and expert systems analysts: What do we know and what do we do?*" *J. Mgmt. Inf. Syst.* **15(1)**, pp. 9–50, Summer 1998, DOI: 10.1080/07421222.1998.11518195.
120. T. R. Shaffer, J. L. Wise, B. M. Walters, S. C. Müller, M. Falcone, and B. Sharif, "*iTrace: Enabling eye tracking on software artifacts within the IDE to support software engineering tasks*". In *ESEC/FSE*, pp. 954–957, Aug 2015, DOI: 10.1145/2786805.2803188.
121. T. M. Shaft and I. Vessey, "*The relevance of application domain knowledge: Characterizing the computer program comprehension process*". *J. Mgmt. Inf. Syst.* **15(1)**, pp. 51–78, 1998, DOI: 10.1080/07421222.1998.11518196.
122. Z. Sharafi, Y. Huang, K. Leach, and W. Weimer, "*Toward an objective measure of developers' cognitive activities*". *ACM Trans. Softw. Eng. & Methodology* **30(3)**, art. 30, Jul 2021, DOI: 10.1145/3434643.

123. Z. Sharafi, B. Sharif, Y.-G. Guéhéneuc, A. Begel, R. Bednarik, and M. Crosby, "*A practical guide on conducting eye tracking studies in software engineering*". *Empirical Softw. Eng.* **25(5)**, pp. 3128–3174, Sep 2020, DOI: 10.1007/s10664-020-09829-4.
124. Z. Sharafi, Z. Soh, and Y.-G. Guéhéneuc, "*A systematic litareture review on the usage of eye-tracking in software engineering*". *Inf. & Softw. Tech.* **67**, pp. 79–107, Nov 2015, DOI: 10.1016/j.infsof.2015.06.008.
125. 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.
126. B. Sharif and J. I. Maletic, "*An eye tracking study on camelCase and under_score identifier styles*". In 18th *Intl. Conf. Program Comprehension*, pp. 196–205, Jun 2010, DOI: 10.1109/ICPC.2010.41.
127. T. Sharma and D. Spinellis, "*A survey of code smells*". *J. Syst. & Softw.* **138**, pp. 158–173, Apr 2018, DOI: 10.1016/j.jss.2017.12.034.
128. B. Shneiderman, "*Measuring computer program quality and comprehension*". *Intl. J. Man-Machine Studies* **9(4)**, pp. 465–478, Jul 1977, DOI: 10.1016/S0020-7373(77)80014-X.
129. 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.
130. F. Shull, J. Singer, and D. I. K. Sjøberg (eds.), *Guide to Advanced Empirical Software Engineering*. Springer-Verlag, 2008.
131. J. Siegmund, "*Program comprehension: Past, present, and future*". In 23rd *Intl. Conf. Softw. Analysis, Evolution, & Reengineering*, pp. 13–20, Mar 2016, DOI: 10.1109/SANER.2016.35.
132. J. Siegmund, C. Kästner, S. Apel, A. Brechmann, and G. Saake, "*Experience from measuring program comprehension—toward a general framework*". In *Software Engineering*, S. Kowalewski and B. Rumpe (eds.), pp. 239–257, Gesellschaft für Informatik e.V., Feb 2013. LNI vol. P-213.
133. J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "*Understanding understanding source code with functional magnetic resonance imaging*". In 36th *Intl. Conf. Softw. Eng.*, pp. 378–389, May 2014, DOI: 10.1145/2568225.2568252.
134. 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.
135. J. Siegmund, N. Peitek, S. Apel, and N. Siegmund, "*Mastering variation in human studies: The role of aggregation*". *ACM Trans. Softw. Eng. & Methodology* **30(1)**, art. 2, Jan 2021, DOI: 10.1145/3406544.
136. J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann, "*Measuring neural efficiency of program comprehension*". In 11th *ESEC/FSE*, pp. 140–150, Sep 2017, DOI: 10.1145/3106237.3106268.
137. 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.
138. H. A. Simon and W. G. Chase, "*Skill in chess*". *American Scientist* **61(4)**, pp. 394–403, Jul-Aug 1973.
139. D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanovic, E. F. Koren, and M. Vokác, "*Conducting realistic experiments in software engineering*". In *Intl. Symp. Empirical Softw. Eng.*, pp. 17–26, Oct 2002, DOI: 10.1109/ISESE.2002.1166921.
140. D. I. K. Sjøberg, B. Anda, E. Arisholm, T. Dybå, M. Jørgensen, A. Karahasanović, and M. Vokáč, "*Challenges and recommendations when increasing the realism of controlled software engineering experiments*". In *Empirical Methods and Studies in Software Engineering: Experiences from ESERNET*, R. Conradi and A. I. Wang (eds.), pp. 24–38, Springer-Verlag, 2003, DOI: 10.1007/978-3-540-45143-3_3. Lect. Notes Comput. Sci. vol. 2765.

141. D. I. K. Sjøberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanović, N.-K. Liborg, and A. C. Rekdal, "*A survey of controlled experiments in software engineering*". *IEEE Trans. Softw. Eng.* **31(9)**, pp. 733–753, Sep 2005, DOI: 10.1109/TSE.2005.97.
142. M. Smith and R. Taffler, "*Readability and understandability: Different measures of the textual complexity of accounting narrative*". *Accounting, Audting & Accountability J.* **5(4)**, pp. 84–98, 1992, DOI: 10.1108/09513579210019549.
143. V. V. Sochat, I. W. Eisenberg, A. Z. Enkavi, J. Li, P. G. Bissett, and R. A. Poldrack, "*The experiment factory: Standardizing behavioral experiments*". *Frontiers in Psychology* **7**, art. 610, Apr 2016, DOI: 10.3389/fpsyg.2016.00610.
144. 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.
145. 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.
146. 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.
147. J. Spolsky, "*The perils of JavaSchools*". www.joelonsoftware.com/2005/12/29/the-perils-of-javaschools-2, 29 Dec 2005.
148. A. Stefik and S. Siebert, "*An empirical investigation into programing language syntax*". *ACM Trans. Computing Education* **13(4)**, art. 19, Nov 2013, DOI: 10.1145/2534973.
149. M.-A. Storey, "*Theories, methods and tools in program comprehension: Past, present and future*". In 13th *IEEE Intl. Workshop Program Comprehension*, 2005, DOI: 10.1109/WPC.2005.38.
150. The National Commission for the Protection of Human Subjects of Biomedical and Behavioral Research, "*The Belmont report*", Apr 1979. URL https://www.hhs.gov/ohrp/regulations-and-policy/belmont-report/read-the-belmont-report/index.html.
151. W. F. Tichy, "*Hints for reviewing empirical work in software engineering*". *Empirical Softw. Eng.* **5(4)**, pp. 309–312, Dec 2000, DOI: 10.1023/A:1009844119158.
152. 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.
153. A. von Mayrhauser and A. M. Vans, "*On the role of hypotheses during opportunistic understanding while porting large scale code*". In 4th *Workshop Program Comprehension*, pp. 68–77, Mar 1996, DOI: 10.1109/WPC.1996.501122.
154. A. von Mayrhauser and A. M. Vans, "*Program understanding behavior during adaptation of large scale software*". In 6th *Workshop Program Comprehension*, pp. 164–172, Jun 1998, DOI: 10.1109/WPC.1998.693345.
155. A. von Mayrhauser, A. M. Vans, and A. E. Howe, "*Program understanding behavior during enhancement of large-scale software*". *J. Softw. Maintenance: Res. & Pract.* **9(5)**, pp. 299–327, Sep/Oct 1997, DOI: 10.1002/(SICI)1096-908X(199709/10)9:5<299::AID-SMR157>3.0.CO;2-S.
156. M. Weiser and J. Shertz, "*Programming problem representation in novice and expert programmers*". *Intl. J. Man-Machine Studies* **19(4)**, pp. 391–398, Oct 1983, DOI: 10.1016/S0020-7373(83)80061-3.
157. L. Weissman, "*Psychological complexity of computer programs: An experimental methodology*". *SIGPLAN Notices* **9(6)**, pp. 25–36, Jun 1974, DOI: 10.1145/953233.953237.
158. L. A. Wilson, Y. Senin, Y. Wang, and V. Rajlich, "*Empirical study of phased model of software change*", Apr 2019. ArXiv:1904:05842 [cs.SE].
159. C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer-Verlag, 2012.