Shorter Identifier Names Take Longer to Comprehend

Johannes Hofmeister
University of Passau, Germany
johannes.hofmeister@uni-passau.de

Janet Siegmund
University of Passau, Germany
janet.siegmund@uni-passau.de

Daniel V. Holt
Heidelberg University, Germany
daniel.holt@psychologie.uni-heidelberg.de

***Abstract*—Developers spend the majority of their time comprehending code, a process in which identifier names play a key role. Although many identifier naming styles exist, they often lack an empirical basis and it is not quite clear whether short or long identifier names facilitate comprehension. In this paper, we investigate the effect of different identifier naming styles (letters, abbreviations, words) on program comprehension, and whether these effects arise because of their length or their semantics. We conducted an experimental study with 72 professional C# developers, who looked for defects in source-code snippets. We used a within-subjects design, such that each developer saw all three versions of identifier naming styles and we measured the time it took them to find a defect. We found that words lead to, on average, 19% faster comprehension speed compared to letters and abbreviations, but we did not find a significant difference in speed between letters and abbreviations. The results of our study suggest that defects in code are more difficult to detect when code contains only letters and abbreviations. Words as identifier names facilitate program comprehension and can help to save costs and improve software quality.**

I. Introduction

Identifier names are important for program comprehension. Their relevance has been discussed for more than 30 years now, for example, by Brooks [7], and Soloway and Ehrlich [27], who explained that they serve as *key beacons* to *program plans*, which activate higher level knowledge about the program and facilitate program comprehension.

Identifiers make up major parts of source code: For example, Deissenboeck and Pizka found that identifier tokens account for approximately 33% of the code of Eclipse 3.1.1 [14]. Developers are free to choose identifier names at their own discretion, which can lead to varying results depending on experience, skill, and mood [26]. In most modern programming languages, identifier names underly only few syntactic limitations, and the actual word can be chosen arbitrarily. For example, developers might be inclined to use single letters as identifier names to save typing effort. When the chosen identifier names are meaningless, developers are most likely slower in comprehending the program's functionality, especially when they are unfamiliar with the code [27]. To alleviate this well-known problem, communities of many programming languages promote style guides, and companies establish specific naming conventions, all with the goal to improve understandability and maintainability of source code. Unfortunately, style guides and conventions only scratch the surface. For example, most style guides prominently encourage the use of a particular separation style for compound identifier names and other structural properties. However, especially semantic properties are often left unmentioned or are limited to the type of word (e.g., classes should have nouns as names [1]). Furthermore, conventions often lack a sound empirical basis [29]. Thus, it is quite unclear how important the influence of identifier naming on comprehensibility and maintainability really is. Understanding the individual aspects of identifier naming can help to choose a particular naming style that optimally supports program comprehension, which in turn can help to improve productivity, and enhance software quality and reduce cost.

Ideally, an identifier name designates a concept from the problem domain, but it does not have to. For example, if a class is named `DataInfoContainer`, but it represents a customer of a shopping site, this makes it difficult to deduce that it actually carries a person's information, because the name is unrelated to the concept (i.e., that it contains data of a customer). This problem has been addressed by other authors: For example, Deissenboeck and Pizka classified bad identifier names along the dimensions of *correctness* and *consistency* [14]. From this viewpoint, an identifier named `DataInfoContainer` is *incorrect*, because its name is neither a subordinate nor superordinate name of the designated concept *person*. Deissenboeck and Pizka's work provides an excellent meta-linguistic framework to discuss the issue, but does not evaluate empirically how inconsistent or incorrect identifier names actually affect developers. Only few authors provide empirical evaluations of the actual impact of names on comprehensibility and maintainability.

In this paper, we describe an experimental study, in which we quantified the impact of *length* and *semantics*[1] of identifier names on program comprehension. To this end, we invited professional software developers to find defects in code that followed different identifier naming conventions (i.e., single letters, abbreviations, words).

We make two contributions in this paper:

- Empirical evidence that full words as identifiers positively influence maintainability and comprehensibility of source code
- A replication package of our experiment and data to help other researchers confirm and extend our results[2]

In the following section, we explain why both single letters and words can have a positive effect on program comprehension. In Section III, we derive and explain our research

[1]We use the term *semantics* to refer to the *meaning* of words in the linguistic sense, rather than a token's semantics in the context of a programming languages, for example, the behavior of the ++ operator.

[2]http://brains-on-code.org/

hypotheses. Section IV describes the experimental setup. We report our findings in Section V and discuss our results in Section VI. We address threats to validity in Section VII.

II. Word-length and Semantics

In this paper, we focus on two main aspects of identifier names, that is, *length* and *semantics*. On the one hand, developers might optimize their code for brevity and choose short identifier names, because they want to reduce typing effort, or because an algorithm is implemented close to a mathematical equation. Abbreviations are also common, resulting in shortened identifier names, for example `configuration` can be shortened to `config`, or `cfg`. On the other hand, identifiers should convey the concept they represent as clearly as possible [3], which is best achieved by using words that actually represent the concept (e.g., a customer is best represented by an identifier named `customer`, not `data`). Words are longer than abbreviations, but their meaning is clearer. Optimizing for either style should not be left to chance or personal preference, but should be based on empirical data with focus on human developers.

Psychological research has long been studying readability and comprehensibility of natural-language texts, and we can find results supporting both short strings and full words as identifier names: On the one hand, very short identifier names, such as abbreviations and letters, require less effort to process cognitively during program comprehension than longer ones, as predicted by the *word-length effect* [4]. This effect describes that lists of short strings are easier to remember than lists of long strings. Thus, developers' performance could be positively affected by very short identifier names, because more items fit into working memory, which helps developers to keep a better overview of the code.

Further findings indicate that the length interacts with the cognitive processing of text. Longer strings take longer to pronounce [5] and have a higher naming latency (i.e., are uttered with delay) than shorter strings [30]. This affects non-words, such as arbitrary strings or nonexistent words (e.g., "awek", "enemenemoo"), as well as low-frequency words (e.g. "penultimate", or "hypochondriasis"), but not common words (e.g., "awake", "hat"), which is called the *word-frequency effect*. Studies that controlled for the ease of articulation showed that not the process of articulation, but rather the required synthesis of the string's phonetics is responsible for the delay [30].

The Dual Route Cascade Model (DRC, [12]) explains these findings: Words that are common in natural language (e.g., "awake", "hat") are stored in a *mental lexicon* (i.e., a dictionary that maps concept to read or spoken words). When they are perceived they can be accessed via a *lexical route* (i.e., they activate a concept or meaning from the mental lexicon). Words that do not exist in this mental lexicon cannot be immediately accessed, because their phonetics have to be synthesized on the fly, thus activating a *phonetic-graphemic route*, a process that is serial in its nature and therefore depends on word length.

In contrast to the word-length effect (shorter words are easier to remember), the *word-superiority effect* predicts that normal words can positively affect program comprehension [23], [28]. Psychological research on the word-superiority effect shows that single letters are easier to detect when they are embedded in words, as opposed to non-words or on their own. The presence of words might facilitate the processing of source code in a similar way and support its comprehension.

When comprehending source code, working memory plays a crucial role, because developers have to work with several programming constructs (e.g., variables, methods). Human working-memory is limited[3], and a word's semantics help relieve cognitive resources through a process called *chunking* [4], in which items are regrouped to more meaningful units. As an example, consider the following method:

```
drawCell(a, b, c, d, e, f)
```

It accepts six floating point value parameters. It is unclear what the variables represent and it is difficult to deduce what they are used for. The method can be refactored to:

```
drawCell(x, y, size, r, g, b)
```

The method still accepts six individual parameters, but their semantics ease reorganizing them in the following manner:

```
drawCell(point, size, color)
```

Thus, instead of six, developers can work with three parameters. Such changes in code are analogous to the mental chunking process and might help relieve working memory.

Additionally, a word's semantics influence how adjacent words are processed, an effect that is called *semantic priming* [11]. This effect may play a role in discovering inconsistencies. When code is too abstract, it might become more difficult to detect semantic defects, as illustrated in Listings 1 and 2:

```
v(u, p):
    u1 = d.u()
    p1 = d.p()
    return u == u1 and p == u1
```

Listing 1: A login function using letter identifier names

```
login(username, password):
    user = db.username()
    pass = db.password()
    return username == user and password == user
```

Listing 2: The same function using word identifier names

The codes are equivalent in their structure; only the identifier names have changed. Both codes are syntactically valid, but the inconsistency is (arguably) easier to detect in Listing 2, when more concrete contextual semantics are present. Since Listing 1 uses only abstract identifier names, it is difficult to detect the semantic defect. However, refactoring the code snippet as in Listing 2 reveals that the wrong comparison is made (`password` against `user`). Thus, meaningful, full word identifier names activate context semantics, which allow developers to evaluate code against its purpose.

To summarize, shortness and semantics of identifier names seem to contradict each other regarding their effect on program

[3]Miller [20] originally argued for a capacity limit of about 7 ± 2 items, while newer research shows that working memory capacity is likely limited to 3 to 5 items [13].

comprehension, and it is unclear from which effect developers could benefit more.

III. Hypotheses

In this study, we address the following research question:

How do identifier name length and semantics affect developers' performance during program comprehension?

We reasoned about possible answers to this question: If program comprehension benefits most from an identifier name's *semantics*, then comprehension of code using words as identifier names, such as `customer` or `request`, should be faster than code using non-words as identifier names, such as abbreviations (e.g., `cus` or `req`), or letters (e.g., `c` or `r`). If an identifier name's *length* is in fact more important for comprehension than its semantics, then shorter (but potentially less meaningful) identifier names should make code faster to understand than code using words as identifier names.

However, determining the actual impact of length is difficult, because different predictions can be drawn from the aforementioned psychological effects. On the one hand, the processing of non-words depends on their length. Thus, abbreviations, compared to single letters should require an increased effort to process, as they are longer than single letters, which should decrease program comprehension performance. On the other hand, the residual semantic properties of abbreviations might facilitate higher cognitive processes (i.e., ease of lexical access, semantic priming, chunking) and therefore, abbreviations should lead to faster comprehension than code with letters as identifier names. Still, it can be expected that the meaning of words is accessed lexically, so abbreviations should still be slower to comprehend than full words.

Further, we reasoned that identifier names' semantic properties affect *program comprehension*, rather than the perceptual processing of code. Therefore, the performance of working with code without trying to comprehend it should not be positively influenced by the presence of full words. However, it is possible, that, on such a low-level of processing, the reduced amount of code leads to faster processing. When the meaning of an identifier is irrelevant (e.g., to find a missing semicolon), then fewer characters imply less code to read and thus could even improve the performance in such tasks.

To answer our research question, we designed a controlled experiment, in which we evaluated the following hypotheses:

- $RH_{Words:Non\text{-}Words}$: Words as identifier names lead to faster comprehension of source code compared to non-words.
- $RH_{Abbreviations:Letters}$: Abbreviations as identifier names lead to faster comprehension of source code compared to single letters.
- RH_{Syntax}: Identifier naming style has no effect on locating syntactical errors

A. Independent variables

To test the relationship between semantics and length, we tested comprehension performance in three conditions: letters, abbreviations, and words. *Letters* are the smallest possible identifier names (an identifier cannot be shorter than one character). *Words* carry more semantics, but also have an increased length. *Abbreviations* form a compromise between these two.

B. Dependent variables

We operationalized the *performance of comprehension* by measuring how long developers investigated a snippet of code until they had found a defect. We assumed that code can only be corrected when it is understood, because developers cannot evaluate the consequences of their changes otherwise. We required developers to indicate when they had found a semantic defect to approximate the exact moment of comprehension.

As a control condition, we also tested whether identifier naming style affects the performance in tasks in which no deep understanding of the code is required. This allowed us to evaluate whether our conditions actually interfered with program comprehension, or whether some other process is being measured. To accomplish this, we measured how much time developers needed to locate a syntax error. Syntax errors, such as missing brackets or semicolons, render the code invalid and require no deep understanding of its identifier names' meanings to be corrected.

IV. Study Design

We tested our hypotheses in a web-based experimental study. Participants were asked to find and correct a defect in six snippets of code. We measured how much time they spent on the task. The scope of our experiment was to *analyze identifier naming styles* for the purpose of *quantifying their effect on program comprehension* with respect to *participants' comprehension speed* in the context of *C# development*. An overview of the experiment following the template provided by Wohlin et al. [33] can be found in Table I.

A. Participants

We recruited 72 professional C# developers (age $M \pm SD$: 35.3 ± 6.8 years). The overall programming experience was 14.0 ± 5.8 years. Their experience with C# was 7.8 ± 3.6 years. We invited them to our experiment via online platforms, such as Twitter and Xing. Additionally, we approached participants of German technology industry conferences.

Overall, 221 people started to participate in the experiment and 135 completed it. The records of 63 participants were removed after applying exclusion criteria to ensure high data quality (see below). The remaining participants were randomly assigned to the experimental sequences. Their details are displayed in Table II.

To obtain the data, we implemented a web application, which is available at the project's website. Since we conducted the experiment online, we could not guarantee an undisturbed working environment. To reduce this potential threat to validity, we controlled our data for disturbances and other factors that could affect their validity. For example, participants were required to have sufficient natural language skills to understand instructions and code comments and had to have a minimum

TABLE I: Experiment overview

Goal	Study the impact of identifier names on program comprehension
Independent Variable	Identifier naming style (Word, Abbreviation, Letter) Task (Semantic Defect, Syntax Error)
Task	Identify semantic defect
Control	Identify syntax error
Dependent Variables	Time to find defect
Secondary Factors	Visual Attention, Correctness
Confounding Factors	Materials, inter-individual differences, item order
Design	Within-subjects

TABLE II: Descriptive participant data

	Category	Percent
School	Abitur / A-Level [2]	81%
	Mittlere Reife / GCSE [3]	11%
	Other	8%
Higher Education	Master's	42%
	Bachelor's	18%
	Vocational training	18%
	No higher ed.	15%
	Other	7%
Employment Status	Employed	81%
	Freelance	17%
	Student	2%
Job Description	Software Developer	51%
	Consultant	12.5%
	Software Engineer	12.5%
	Project Manager	10%
	Software Architect	8%
	Other	6%

[2]Equivalent to 12 to 13 years of schooling
[3]Equivalent to 10 years of schooling

of one year experience with C# in practical use. They provided self-ratings of their language proficiency on a scale from 1 to 6 for German and English. Since the code was written using English identifier names and the instructions were given in German, the data of participants with ratings below 4 in either category were excluded. We targeted professional developers and avoided selecting students; however, in the final sample the records of two students remained. They stated appropriate experience with C# to be considered professional developers. All exclusion criteria and number of affected records are listed in Table III.

B. Task

The participants' task was to find one defect in a snippet of code. The task was repeated six times: After they had worked on three snippets finding *semantic defects*, they were asked to work on three more snippets but to look for *syntax errors* now.

To gather coarse-grained data about participants' visual focus, we used an implementation of the restricted focus viewer [16], which also helped us to detect distracted participants. In our implementation, participants' view on the code was limited to 7 lines at once (approximately one third of the complete snippet), but the frame could be shifted up and down using the arrow keys to reveal the rest of the code. We called this feature the *letterbox*, because it mimics spying through the letter slit in a door or mailbox.

When participants had found the defect, they pressed the space bar, freezing the letterbox, and opening a dialog screen, in which participants entered the line number of the defect, its description, and a correction. We measured the time how long participants looked at the code until they indicated that they had found the defect. In other words, from the entire duration of the task, we subtracted the time spent answering the dialog and only evaluated the time that participants interacted with code. This way, we analyzed only the time required to comprehend the code. Participants who had failed to find the defect in a snippet after three attempts were allowed to finish the experiment, but their data was excluded.

We used the time required to find semantic defects as a measure of program comprehension. This is easy to score for correctness and the response has a well-defined time point, allowing for reaction time analyses. Furthermore, finding semantic defects requires that the intentions behind the code (what should it do?) and the semantics of its operation (what does it do?) are understood to give a correct response. Because the study was conducted online, we ruled out think-aloud protocols. Locating defects is a common programming task, which renders it a relevant target for studying program comprehension.

TABLE III: Exclusion criteria

		Criterion	n
Language Level	German (1 - 6)	<4	2
	English (1 - 6)	<4	9
Programming Experience	C# Skill (1 - 5)	<4	24
	C# Experience (Years)	<1	8
Behavior	Encountered distractions?	Yes	17
	Worked on task conscientiously?	No	1
	Attempts to succeed	>3x	15
	Freeze, AFK (No interaction)	>1 min	4
	Too Slow (time per trial)	>10 min	14
Other	Participated in pilot study?	Yes	4
	Participated before?	Yes	8
	Total (Criteria not mutually exclusive)		63

C. Materials

We developed eleven code snippets containing simple algorithms. We created new code to ensure that no participant had seen it before. The snippets needed to be simple enough to be comprehensible in a reasonable time frame, but complex enough for defects to "hide" in the code. Each snippet consisted of a self-contained static function, with a length of 15 lines. Listing 3 shows an example. We limited the code to language features from C# 2.0, such as loops, conditionals, and basic .NET API calls. We avoided more complex structures, such

TABLE IV: Examples of the different identifier naming styles used.

Type	Example
Word	`request`, `histogram`
Abbreviation	`rqs`, `hst`
Letter	`a`, `b`

as recursion, or specific APIs (e.g., Language-Integrated Query (LINQ)), to avoid bias due to extensive C# experience. Each snippet came in three versions, in which the identifier names were altered to either words, abbreviations or letters. Examples are shown in Table IV. Each snippet had one version with a *semantic defect* and a one with a *syntax error*. The errors were placed in similar locations in the code to avoid bias due to different locations of the errors.

Each snippet was built with expressive word identifier names first. From this version, two derived versions were generated by replacing the identifier names in an automated process. Abbreviations were generated by removing vowels from identifier names and leaving the first three remaining consonants in place (e.g., `request` became `rqs`), such that they would still contain traces of the original word. For some words, which are commonly abbreviated using the first three letters (e.g., `len` for `length`), we used these abbreviations instead. In the letter version, identifiers were named alphabetically, in the order of occurrence, ensuring the validity of the code. We decided on alphabetical replacement, to guarantee that the identifiers did not resemble the original identifier names in any way. The standard .NET API was left intact (e.g., identifiers like `List`, `Dictionary` were not abbreviated). Each function was commented on top. The first line contained an explanatory description of the method's desired functionality. The following lines documented the variables and, in the abbreviation and letter versions, showed their original meaning.

We evaluated the snippets' suitability in a pilot study. Participants were shown three snippets with word identifier names, and we measured the time until participants found a semantic defect. The data of the pilot study were not used to answer our research question. We had recruited two different samples of participants online, but could not prevent a slight overlap between the samples. In order to prevent learning effects, records of people who took part in the pilot study were excluded.

We removed five snippets after the pilot study, because the measured times exposed too much variance (i.e. the difference between fast and slow participants was too high), or because they where too difficult (i.e. all participants were comparatively slow). To minimize differences in comprehension performance caused by variances in the snippets, we used six out of eleven snippets for the actual experiment.

Listings 3, 4, and 5 show three snippet versions, all of which show the same algorithm that naïvely concatenates two lists. The defect resides in Line 21 and its correction could be: `result[index + length] = second;`.

```
 1: // ConcatLists: Concatenates two lists of the
                                      same length
 2: // start: collection of elements at the start
 3: // end: collection of elements to append
 4: // length: length
 5: // result: result
 6: // index: index
 7: // first: first
 8: // second: second
 9:
10: public static int[] ConcatLists(int[] start,
                                       int[] end)
11: {
12:     int length = start.Length;
13:     var result = new int[length * 2];
14:
15:     for (int index = 0; index < length; index++)
16:     {
17:         int first = start[index];
18:         int second = end[index];
19:
20:         result[index] = first;
21:         result[index + 1] = second;
22:     }
23:     return result;
24: }
```

Listing 3: Snippet using words as identifier names

```
 1: // Cnc: Concatenates two lists of the same
                                        length
 2: // str: collection of elements at the start
 3: // end: collection of elements to append
 4: // len: length
 5: // rsl: result
 6: // idx: index
 7: // frs: first
 8: // scn: second
 9:
10: public static int[] Cnc(int[] str, int[] end)
11: {
12:     int len = str.Length;
13:     var rsl = new int[len * 2];
14:
15:     for (int idx = 0; idx < len; idx++)
16:     {
17:         int frs = str[idx];
18:         int scn = end[idx];
19:
20:         rsl[idx] = frs;
21:         rsl[idx + 1] = scn;
22:     }
23:     return rsl;
24: }
```

Listing 4: Snippet using abbreviations as identifier names

```
 1: // a: Concatenates two lists of the same length
 2: // b: collection of elements at the start
 3: // c: collection of elements to append
 4: // d: length
 5: // e: result
 6: // f: index
 7: // g: first
 8: // h: second
 9:
10: public static int[] a(int[] b, int[] c)
11: {
12:     int d = b.Length;
13:     var e = new int[d * 2];
14:
15:     for (int f = 0; f < d; f++)
16:     {
17:         int g = b[f];
18:         int h = c[f];
19:
20:         e[f] = g;
21:         e[f + 1] = h;
22:     }
23:     return e;
24: }
```

Listing 5: Snippet using letters as identifier names

D. Procedure

Participants were invited to a public website where they found an introduction text, legal information related to informed consent, and a privacy statement. From there, they entered the experiment, starting with several questionnaires. They were asked to provide information about their education, employment status, and professional experience. They continued with a tutorial that gradually familiarized them with the actual experiment.

After the tutorial, an overview of the upcoming task was presented. On the next screen, participants were instructed to press the space bar to start the trial. The participants inspected the code, searching for a defect. When they had found the defect, they pressed the space bar again, opening the aforementioned correction dialog. After filling out the dialog, participants received feedback whether their answer was correct to motivate them to continue.

After the experiment, a final questionnaire asked for some demographic data, and whether or not they had been distracted during the experiment.

E. Design

The goal of our experiment was to quantify the effect of identifier naming styles on program comprehension. To isolate the effect as much as possible, we controlled several factors that could affect comprehension performance, namely the effects of inter-individual differences between participants, difficulty of the snippets, and order effects. This resulted in the design illustrated in Figure 1.

1) Inter-Individual Differences: To control for inter-individual differences, we used a within subjects-design, such that every participant saw all realizations of the different identifier naming styles. This compensated participants' different skill levels, for example, that slow readers would be slower in every task.

2) Material Effects: To reduce side-effects caused by our materials (e.g., a complex problem takes longer to comprehend, but not because of its identifier naming style), we grouped the snippets into two sets of three snippets each, depending on their difficulty (*Group Easy* and *Group Difficult*) established in the pilot study. In the final experiment, half on the participants were shown three snippets with semantic defects from Group Easy first, followed by three snippets containing syntax errors from Group Difficult, vice versa for the other half of participants. Furthermore, we permuted the order of snippets within each group to counterbalance snippet-specific difficulty and order effects.

3) Effects of Condition Order: To reduce the effects of condition order, which may lead to increased or decreased performance over the course of the experiment, we also permuted the order of identifier naming styles in each group. Table V shows an example trial for one participant, with Group Easy consisting of Snippets 1, 2, and 3, Group Difficult consisting of Snippets 4, 5, and 6.

Combining and permuting these factors, we generated 72 different sequences of snippets (3! identifier naming style $\times$ 3! snippet order $\times$ 2 difficulty order), which also defined the sample size.

TABLE V: Example trial sequence for one participant

Sequence	Group	Task	Snippet	Style
1	Difficult	Semantic Defect	6	Letter
2	Difficult	Semantic Defect	4	Word
3	Difficult	Semantic Defect	5	Abbreviation
4	Easy	Syntax Error	1	Word
5	Easy	Syntax Error	2	Abbreviation
6	Easy	Syntax Error	3	Letter

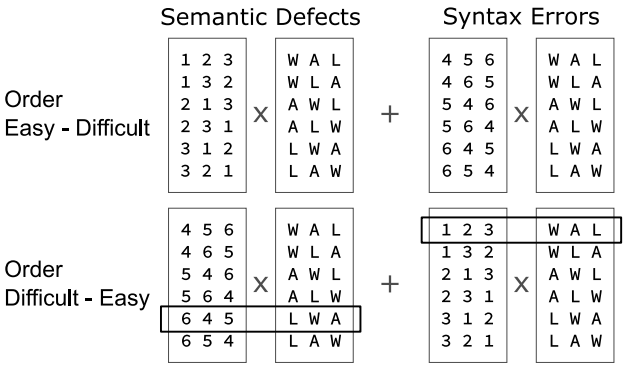


Fig. 1: We used a balanced design to control for learning effects and effects caused by our stimulus material. Snippet Group Easy: 1,2,3. Snippet Group Difficult: 4,5,6. The groups were permuted and multiplied with the identifier type: Word (W), Abbreviation (A), Letter (L). Each ordering contained 36 sets. Each set consisted of 6 trials, for example 6L-4W-5A-1W-2A-3L.

In summary, every participant saw:

- Three semantic defects first, then three syntax errors
- All identifier naming styles
- All six snippets, encountering each snippet only once

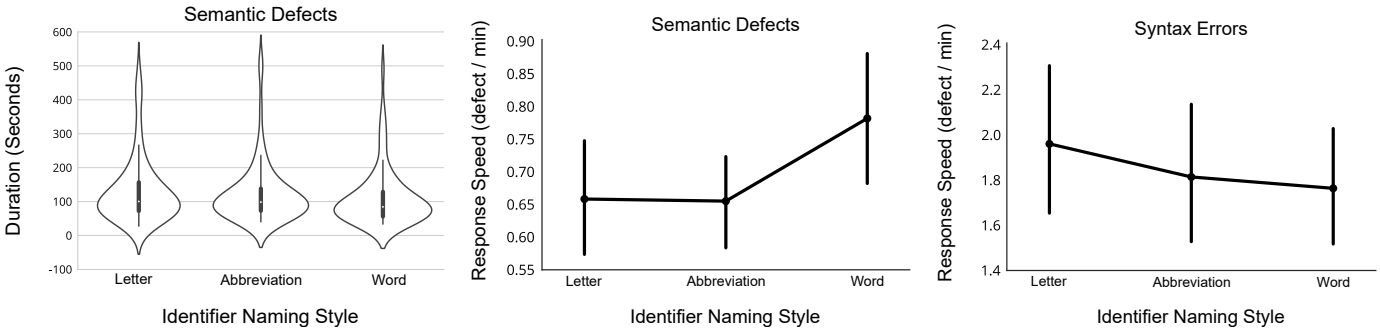
During the pilot study, we had observed that syntax errors where found much faster than semantic defects. Thus, to prevent participants from being discouraged by the upcoming amount of work and drop out of the experiment, we explained that the last three items together (syntax errors) required about as much time as one of the previous items (semantic defects). Altogether, we explained that the total duration of the experiment was 20 to 30 minutes.

V. Results

To test our hypotheses, we analyzed the response time data of participants (i.e., the time they viewed code until they pressed the space bar). In this section, we first present data preparation and the descriptive statistics, and then the test of our hypotheses.

A. Data Preparation and Descriptive Statistics

Table VI and Figure 2a show summaries of the raw reaction time data, split by identifier naming styles. As illustrated in Figure 2a, the data are skewed, such that fast responses accumulate on the left and with a tail of slow outliers on the right side of



(a) Untransformed response times, grouped by identifier naming style. The distributions were skewed. The plots disregard the inter-individual differences which we controlled for in our statistical tests

(b) When looking for semantic defects, participants detected them faster when code contained words as identifier names, in comparison to letters or abbreviations. There was no significant difference between abbreviations and letters. The vertical bars show 95% confidence intervals.

(c) Although there appears to be an effect of identifier names on finding syntax errors, it is not statistically significant. Overall, syntax errors were found much faster. The vertical bars show 95% confidence intervals.

Fig. 2: Effect of identifier naming style on response times and speed

TABLE VI: Duration of interaction with code, split by identifier naming styles. Values show median and interquartile range (IQR) in `mm:ss` (minutes, seconds).

	Semantic		Syntactic	
Type	Median	IQR	Median	IQR
Word	01:24.48	01:12.78	00:39.42	0:49.00
Abbreviation	01:38.57	01:05.37	00:36.71	0:53.92
Letter	01:40.36	01:24.87	00:35.74	0:30.22

the distribution, a phenomenon common for reaction times [22]. Under these circumstances common descriptive statistics, such as mean and standard deviation, become difficult to interpret. In this case, the median as a measure of central tendency and the interquartile range (IQR[4]) as a measure of dispersion are more suitable [31].

The presence of outliers can reduce the power of experimental analyses. According to Ratcliff [22], outliers are "response times generated by processes that are not the ones being studied"; for example, participants could have been distracted, or might have lost attention. There are several ways to reduce the impact of outliers and retain the power of statistical tools, including trimming, winsorizing, and transforming. Trimming removes outlier data points above a certain cutoff threshold, winsorizing replaces them with the threshold value, and transforming the data changes the data distribution, thus prioritizing certain values [22], [19].

We chose to transform the data using an inverse transformation, which represents a good compromise between reducing the impact of outliers, data retention, and interpretability when applied to reaction times [22]. We preferred an inverse transformation over a logarithmic transformation as inversely transformed response time data have an intuitive interpretation.

[4] $IQR = Q_3 - Q_1$; where Q1 (i.e., the first Quartile) contains the 25% of the fastest response times, and the following 3 quartiles the remaining three quarters of the data.

TABLE VII: Response and Comment Speed (data transformed with $^1/_{RT}$) during the semantic and syntax tasks, split by identifier naming style.

		Semantic		Syntactic	
Measure	Type	M	SD	M	SD
Defects / Minute	Word	0.78	0.42	1.76	1.13
	Abbreviation	0.65	0.31	1.81	1.31
	Letter	0.66	0.39	1.96	1.39
Inverted Comment Reading Time	Word	3.27	2.96	17.85	12.66
	Abbreviation	2.64	2.59	14.34	10.75
	Letter	2.33	1.88	14.77	10.37

The transformed values simply express *defects per minute* rather than *minutes per defect*, i.e., the speed of finding defects. The data are displayed in table VII. For example, when finding semantic defects in code with words instead of abbreviations and letters, participants found on average 19% more defects per minute and thus were faster when words were used as identifier names.

B. Hypothesis Testing

We calculated inferential statistics for *semantic defects* and *syntax errors* separately. We concentrated our analysis on the semantic task and used more economical methods for the analysis of the syntax task, because the syntax task was designed as control condition and a sanity check for our main hypotheses. We define a significance level of $\alpha = .05$ for all tests.

1) Semantic Defects: We tested the effect of identifier naming style on semantic defects using linear contrasts [32]. Analysis of variance (ANOVA) approaches use an omnibus F-test to establish whether there is an overall effect of an experimental factor (e.g., identifier naming style), but without exactly locating the effect [2]. Linear contrasts test groups of factor levels (e.g., words and non-words) against each other, which allows to test more specific hypotheses. As explained in Section

II, we had assumptions about the relationship between words, abbreviations and letters, and thus were able to formulate a priori linear contrasts. The test creates a weighted contrast variable, which is then tested with a Student's t-test [32]. As the inverse transformation has a normalizing effect on reaction time data and t-tests are robust against small deviations from distributional assumptions when the sample size is sufficiently large, deviations from normality are not a problematic issue for the present analyses.

We compared *words* against *non-words*, by grouping letters and abbreviations in a contrast comparison ($\Psi_{\text{Words:Non-Words}}$). We then compared *letters* against *abbreviations* and omitted words ($\Psi_{\text{Letters:Abbreviations}}$). Aligned with our statistical hypotheses, we tested:

$$\Psi_{\text{Words:Non-Words}} : \text{Performance}_w > (\text{Performance}_a + \text{Performance}_l)$$
$$\Psi_{\text{Letters:Abbreviations}} : \text{Performance}_a > \text{Performance}_l$$

With Performance : Comprehension Performance Speed as $^1/_{RT}$, where higher values indicates better performance.

We found a statistically significant difference between comprehension speed for words and non-words ($t_{\Psi\text{Words:Non-Words}}(71) = 2.73$; $p = .004$). There was no significant difference between letters and abbreviations ($t_{\Psi\text{Letters:Abbreviations}}(71) = 0.07$; n.s.). The difference between words and non-words indicated a small to medium-sized effect ($d_z = 0.32$, [10]). The effect is illustrated in Figure 2b.

2) Syntax Errors: We had reasoned that identifier naming style would not influence locating syntactical errors, and thus used the more economical omnibus ANOVA approach, rather than specifying contrasts for the effect. We could not find a significant effect of identifier naming styles on the detection of syntax errors, as illustrated in Figure 2c, ($F(2, 142) = 0.8$; n.s.). Considering that our study has 80% statistical power to detect effect sizes as small as $\eta^2 = .04$, we interpret this result as support for the assumption that identifier names have at most a negligible effect on finding syntax errors.

C. Visual Attention

Additionally, we analyzed the visual attention data, which were obtained with the letterbox. The identifier naming style affected how much time participants spent reading (and re-reading) the comments at the beginning of the snippet, see Table VII ($F_{\text{Sem}}(2, 142) = 5.35$; $p = .006$; $\eta^2_{\text{Sem}\langle\text{Style}\rangle} = 0.07$; $F_{\text{Syn}}(2, 142) = 3.60$; $p = .03$; $\eta^2_{\text{Syn}\langle\text{Style}\rangle} = 0.05$). This affected the semantic and syntactic tasks equally.

VI. Discussion

To summarize the results, our data show that participants found semantic defects significantly faster when the presented code used normal words as identifier names, compared to non-words (i.e., abbreviations and letters). When code used abbreviations or letters as identifier names, participants were equally fast. Finding syntax errors appears to be unaffected by identifier naming style.

These results indicate that program comprehension benefits from explicit identifier names, as comprehension of code with words as identifier names was on average 19% faster, compared to abbreviations and letters.

Although the *word-length effect* predicts that shorter strings are easier to remember, letters and abbreviations did not lead to an improvement of comprehension performance. Instead, longer words seemed to facilitate comprehension, as semantic defects were discovered faster when words were used as identifier names. Moreover, contrasting the semantic tasks with the syntactic tasks shows that purely perceptual properties of identifier names, such as their length, are also insufficient to explain performance differences even when the task does not require semantic judgments. However, in this case the word identifiers' semantic content is not a benefit either. The observed differences in comprehension performance are likely caused by the words' semantic content, which facilitates cognitive processes of developers, such as *chunking* or *semantic priming*, and generally relieves working memory. This is also supported by the fact that in the word condition, participants scrolled less frequently to the comments of the source code to retrieve the meaning of a variable.

In other words, word identifier names allow developers to access the meaning of a concept represented by an identifier directly, which may allow them to reason about the code more easily. However, to distinguish the precise nature of these processes, further experiments are necessary.

Considering these results, arguments in favor of non-words seem questionable. For example, the disadvantage of increased typing effort of longer words is outweighed by the benefits of their increased semantics, especially when considering that code is more often read than written. In practice, this drawback could be diminished with appropriate tooling. For example, modern IDEs provide auto-completion facilities, which already reduce typing effort. In future studies, it would be interesting to quantify the relationship between typing effort in modern IDEs and comprehensibility.

To summarize, our data indicate that using words as identifier names benefits comprehensibility and maintainability of source code. Thus, we provide evidence that developers should follow this convention, because it will most likely result in comprehensible and maintainable code, which in turn has an effect on code quality.

VII. Threats to validity

A. Internal validity

We used only 33% of all obtained data records, which could be interpreted as a sampling bias, but was in fact a result of our strict filtering rules that were applied to improve the quality of our data by reducing the effect of nuisance factors (e.g., language barriers, distractions).

B. Construct validity

Another threat is raised by our operationalization of program comprehension as "time to find a defect". Possibly, the time to

find a defect has a certain overhead, compared to comprehension in isolation, and may not be a fully equivalent measure for program comprehension performance. However, we expect a large overlap between these two constructs. For example, identifying that something "is a car" is arguably an easier task than to identify the reason why it will not start, but the former process is required to enable the latter. Similarly, identifying that a code snippet sorts an array is a different task than to find out the complexity of the algorithm, or whether or not it contains a defective corner-case.

C. External validity

We limited our sampling criteria to a very specific population, namely German professional C# developers. Thus, our findings should not be overgeneralized. Only two participants identified as female in the initial raw data, but their records were not included in the final dataset after applying exclusion criteria. However, program comprehension is a complex cognitive activity, and we expect that professional education and experience would outweigh potential gender differences.

Our stimulus materials were limited to procedural, algorithmic problems and therefore do not allow us to draw conclusions about the impact of identifier naming styles in complex, object-oriented environments, where identifiers like `AbstractSingletonProxyFactoryBean` can be common. These very long identifier names try to be explicit, but they might be too abstract to provide meaningful semantics to facilitate program comprehension, thus hindering the performance of program comprehension. Also, the word-length effect might play a stronger role here. However, this conclusion needs to be tested in following experiments.

In our study, we instructed participants to search for a semantic defect somewhere in the code. These instructions may have activated additional resources for the comprehension of code, such as knowledge about defects, search strategies, or direction of attention. Under normal circumstances, such priming does not necessarily occur, and developers might be required to identify whether there exists a problem in the code at all, and it is possible that comprehending code without such hints is a more tedious process. Thus, our instructions might have influenced our measures of performance by activating additional resources that would otherwise not affect the comprehension process, thus reducing the size of the effect of identifier naming style. Without additional instructions, words as identifier names with rich semantics might be even more valuable.

For our experimental setup, we chose tasks that were relevant to developers' daily work (i.e., finding semantic defects), but in order to reach out to participants, these tasks were performed online, rather than in an actual IDE, and thus might be regarded as artificial. Our web-application might have slowed down comprehension performance by means of the restricted focus viewer, and comprehension should be faster under normal circumstances. Because this impediment was present to all participants in all experimental conditions, it may have normalized the response times and helped to accentuate the effect of identifier naming styles, resulting in an improved power of our tools.

Finally, it should be noted that modern, complex code bases can have millions of lines of code. The observed effect size of $d_z = 0.32$ indicates a small to medium-sized effect, which we observed in only 15 lines of code. We suspect the effect of identifier naming styles to be even more pronounced in larger programs. Participants were, on average, 19% faster in comprehending 15 lines of code with word identifier names, which could save hours or even weeks of time required to comprehend code, when more code must be investigated. However, it is difficult to predict this effect with certainty: It is possible that common abbreviations in a larger code base are memorized quickly, and developers do not suffer a penalty during program comprehension. For example, the use of the letter `i` as an index in a `for`-loop can be considered idiomatic. A similar effect might be found for a company's domain-specific identifier names.

Although we controlled many factors, further studies are necessary to fully understand the effect. Ideally, the effect can be isolated by conducting the study in a laboratory setting, or possibly with stimulus material written in other programming languages.

VIII. Related Work

Lawrie and others performed a similar experiment, in which participants read code with letters, abbreviations, and words [17]. They found that responses for identifier names using abbreviations showed similarities to responses for normal words, whereas in our experiment, they were more similar to responses for letters. However, our findings are congruent with theirs, in that words as identifier names lead to a better comprehension compared to letters. We attribute this difference to the strategies used for building abbreviations. Lawrie and others abbreviated longer composite identifier names rather than single words (e.g., `isPrimeNumber` to `isPriNum`). Such abbreviations retain more similarities to the original identifier name compared to the identifiers used in our study. Thus, we see their results in accordance with ours and understand them as and extension of our hypotheses.

In a subsequent paper, Lawrie and others found indications that identifier name length interacts with working memory to such an extent that words and abbreviations are easier to identify in recognition tasks than single letters [18]. Again, this applies to the longer abbreviations in their study. In these results, we find further confirmation for our results, that semantics are relevant for the comprehension of source code and that purely perceptual explanations of differences in comprehension performance are insufficient.

Epelboim [15] conducted a study of identifier separation styles, a topic that was also addressed by Binkley and others [6], as well as Sharif and Maletic [25], who replicated the study of Binkley and others [6]. All these studies agree that `under_scores` in identifier names facilitate program comprehension compared to `camelCase`. Sharif and Maletic [25] emphasize the influence of familiarity with a style in this regard, showing that novices benefit from underscores more than experts.

Although we found that semantic properties are more important for program comprehension than low-level perceptual properties of identifier names, the research on identifier-splitting techniques shows that both aspects should be considered as complementary. Syntactic properties of identifier names may facilitate perceptual processes, whereas semantic properties facilitate higher level cognitive processes. Both aspects should be considered to write code that people can understand optimally.

Further, both the studies by Sharif and Maletic, and Binkley and others found that better comprehension was achieved when participants were presented with code that was congruent with their previous experiences. The study by Binkley and others showed that participants who were experienced with `camelCase` took less time to identify `camelCase` identifiers compared to `under_score` identifiers. Aligning their own results with these findings, Sharif and Maletic conclude that "with more experience (training), the effect of identifier style on performance is reduced, but not eliminated" [25]. Thus, experience seems to play a role when determining relevant factors of program comprehension. Our data exhibit similar characteristics. We found that the observed effects (the impact of identifier naming styles) reside in the middle and the tail of the distribution of reaction time data. This indicates that *experts* (i.e., the fastest developers in our sample) were less influenced by shorter or abbreviated identifier names than developers with average performances.

Ceccato and others [9] analyzed different code-obfuscation techniques that intentionally make code difficult to comprehend. They could show that renaming identifiers to single characters is an effective obfuscation technique to hinder program comprehension, although it does not render it impossible. These results underline the importance of good identifier naming styles, as comprehending code seems to be easier when words are used and impeded when letters are present.

Scalabrino and others found that using 'textual' properties of source code, including semantic aspects, such as coherence and narrowness of identifier names, improve the prediction of *readability* over and above using structural aspects such as line lengths, number of identifiers, or number of parentheses [24]. In the work of Scalabrino, as well as other code readability studies (e.g., [8], [21]), readability is often operationalized as participants' subjective judgments whether or not a snippet is readable. In contrast to subjective ratings such as these, our experiment employed response times as a behavioral performance measure. Ideally, subjective and objective measures of readability will lead to converging results, but this relationship should be investigated further to clearly define the validity of either construct.

IX. Conclusion

Given that maintenance and program comprehension play a crucial role in software development (most likely more than actually typing code), it seems advisable to use explicit full-word identifiers. Our results indicate that abbreviations and letters reduce a program's comprehensibility, and their presence might be an indicator for lower quality code.

We could show that shorter identifier names are not necessarily better, because words' semantic properties of identifier names enable cognitive processes that facilitate comprehension. Developers should optimize their code to support these cognitive processes by using explicit identifier names.

To facilitate comprehension as much as possible, appropriate rules for style guides should consider perceptual and semantic properties, and discourage the use of abbreviations and letters, and encourage the use of explicit, clear identifier names - given that they wish to improve the quality of software, and reduce its development and maintenance costs.

In future work, we intend to replicate the study with an eye-tracker to gain more data about participants' eye movements. Additionally, we also plan to recruit participants from different populations. Specifically, beginning programmers are an interesting group to understand the effect of words designating the implemented concept on program comprehension. In general, we welcome replications of the results presented here to evaluate their robustness in different contexts. For this purpose, we provide a replication package with all required materials on the project website (see Footnote 2).

Acknowledgments

This work has been supported by the DFG grant SI 2045/2-1.

References

[1] Class Naming Guidelines [online]. available: https://msdn.microsoft.com/en-us/library/4xhs4564(v=vs.71).aspx.

[2] T. Anderson and J. D. Finn. *The New Statistical Analysis of Data*. Springer, New York, NY, USA, 1996.

[3] N. Anquetil and T. Lethbridge. Assessing the Relevance of Identifier Names in a Legacy Software System. In *Conf. Centre for Advanced Studies on Collaborative Research*, CASCON '98, pages 1–10, Toronto, Ontario, Canada, 1998. IBM Press.

[4] A. D. Baddeley, N. Thomson, and M. Buchanan. Word Length and the Structure of Short-Term Memory. *Journal of Verbal Learning and Verbal Behavior*, 14(6):575 – 589, 1975.

[5] D. A. Balota and J. I. Chumbley. The Locus of Word-Frequency Effects in the Pronunciation Task: Lexical Access and/or Production? *Journal of Memory and Language*, 24(1):89 – 106, 1985.

[6] D. Binkley, M. Davis, D. Lawrie, and C. Morrell. To CamelCase or under_score. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 158–167, May 2009.

[7] R. Brooks. Towards a Theory of the Comprehension of Computer Programs. *Int.ĺ Journal of Man-Machine Studies*, 18(6):543 – 554, 1983.

[8] R. P. L. Buse and W. R. Weimer. Learning a Metric for Code Readability. *IEEE Trans. Softw. Eng. (TSE)*, 36(4):546–558, July 2010.

[9] 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.*, 19:1040–1074, 2014.

[10] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Erlbaum, Hillsdale, NJ, 1988.

[11] A. M. Collins and E. F. Loftus. A Spreading-Activation Theory of Semantic Processing. *Psychological Review*, 82(6):407–428, Nov. 1975.

[12] M. Coltheart, K. Rastle, C. Perry, R. Langdon, and J. Ziegler. DRC: A Dual Route Cascaded Model of Visual Word Recognition and Reading Aloud. *Psychological Review*, 108(1):204 – 256, 2001.

[13] N. Cowan. The Magical Number 4 in Short-Term Memory: A Reconsideration of Mental Storage Capacity. *Behavioral and Brain Sciences*, 24(1):87 – 185, 2001.

[14] F. Deissenboeck and M. Pizka. Concise and Consistent Naming. *Software Quality Control*, 14(3):261–282, Sept. 2006.

[15] J. Epelboim, J. R. Booth, R. Ashkenazy, A. Taleghani, and R. M. Steinman. Fillers and Spaces in Text: The Importance of Word Recognition During Reading. *Vision Research*, 37(20):2899–2914, Oct. 1997.

[16] A. R. Jansen, A. F. Blackwell, and K. Marriott. A Tool for Tracking Visual Attention: The Restricted Focus Viewer. *Behavior Research Methods, Instruments, & Computers*, 35(1):57–69, 2003.

[17] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a Name? A Study of Identifiers. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 3–12, June 2006.

[18] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. Effective Identifier Names for Comprehension and Memory. *Innovations in Systems and Software Engineering*, 3(4):303–318, 2007.

[19] R. Leonhart. *Lehrbuch Statistik. Einstieg und Vertiefung*. Hans Huber, Hogrefe AG, Bern, 2nd edition, 2009.

[20] G. A. Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information. *Psychological Review*, 101(2):343 – 352, 1994.

[21] D. Posnett, A. Hindle, and P. Devanbu. A Simpler Model of Software Readability. In *Proc. Working Conf. on Mining Software Repositories (MSR)*, pages 73–82, New York, NY, USA, 2011. ACM.

[22] R. Ratcliff. Methods for Dealing With Reaction Time Outliers. *Psychological Bulletin*, 114(3):510–532, Nov. 1993.

[23] G. M. Reicher. Perceptual Recognition as a Function of Meaningfulness of Stimulus Material. *Journal of Experimental Psychology*, 81(2):275–280, Aug. 1969.

[24] S. Scalabrino, M. Linares-Vsquez, D. Poshyvanyk, and R. Oliveto. Improving code readability models with textual features. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 1–10, May 2016.

[25] B. Sharif and J. I. Maletic. An Eye Tracking Study on camelCase and Under_score Identifier Styles. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, Proc. Int'l Conf. Program Comprehension (ICPC), pages 196–205, Washington, DC, USA, 2010. IEEE Computer Society.

[26] H. Sneed. Object-oriented COBOL Recycling. In *Reverse Engineering, 1996., Proceedings of the Third Working Conference on*, pages 169–178, Nov. 1996.

[27] E. Soloway and K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.*, SE-10(5):595–609, Sept. 1984.

[28] K. Spalek. Wortverarbeitung. In B. Hhle, editor, *Psycholinguistik*, pages 68–80. Akademie Verlag, Berlin, 1. edition, 2010.

[29] W. F. Tichy. Should Computer Scientists Experiment More? In *IEEE Computer*, 1998.

[30] B. S. Weekes. Differential Effects of Number of Letters on Word and Nonword Naming Latency. *The Quarterly Journal of Experimental Psychology Section A*, 50(2):439–456, May 1997.

[31] R. Whelan. Effective Analysis of Reaction Time Data. *The Psychological Record*, 58(3):475, 2008.

[32] T. Wickens and G. Keppel. *Design and Analysis: A Researcher's Handbook*. Prentice Hall, Upper Saddle River, N.J., 4th international edition edition, July 2004.

[33] C. Wohlin, P. Runeson, M. Hst, M. C. Ohlsson, B. Regnell, and A. Wessln. *Experimentation in Software Engineering*. Springer, Berlin, Heidelberg, 2012.