Meaningful Identifier Names: The Case of Single-Letter Variables

Gal Beniamini Sarah Gingichashvili Alon Klein Orbach Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem, 91904 Jerusalem, Israel

***Abstract*—It is widely accepted that variable names in computer programs should be meaningful, and that this aids program comprehension. "Meaningful" is commonly interpreted as favoring long descriptive names. However, there is at least some use of short and even single-letter names: using `i` in loops is very common, and we show (by extracting variable names from 1000 popular github projects in 5 languages) that some other letters are also widely used. In addition, controlled experiments with different versions of the same functions (specifically, different variable names) failed to show significant differences in ability to modify the code. Finally, an online survey showed that certain letters are strongly associated with certain types and meanings. This implies that a single letter can in fact convey meaning. The conclusion from all this is that single letter variables can indeed be used beneficially in certain cases, leading to more concise code.**

***Keywords*—Program comprehension, meaningful identifier names, single-letter names**

I. Introduction

Giving identifiers in computer programs meaningful names is a widely accepted best practice. This is reflected both in coding standards and in programming teaching materials. For example, the Google C++ Style Guide states "Give as descriptive a name as possible, within reason. Do not worry about saving horizontal space as it is far more important to make your code immediately understandable by a new reader. Do not use abbreviations that are ambiguous or unfamiliar to readers outside your project, and do not abbreviate by deleting letters within a word."[1] Likewise, the Free Software Foundation's GNU Coding Standards say "The names of global variables and functions in a program serve as comments of a sort. So don't choose terse names—instead, look for names that give useful information about the meaning of the variable or function."[2] McConnell devotes all of Chapter 11 of the second edition of *Code Complete* to variable names [17]. This includes the following: "The most important consideration in naming a variable is that the name fully and accurately describe the entity the variable represents. An effective technique for coming up with a good name is to state in words what the variable represents. Often that statement itself is the best variable name."

As these three examples demonstrate, a recurring theme in variable naming is the suggestion that long descriptive names should be used. But on the other hand, one often finds loops indexed by a variable named `i`, rather than the more descriptive `indexOfLoopOverAllRecords` or something to that effect. This raises the question of whether this supposed violation of the best practice has measurable consequences, or maybe it is actually acceptable.

More precisely, one may ask *under what circumstances* is it legitimate to use abbreviated or even single-letter variable names. For example, we may speculate that this can be appropriate when variables have a limited local scope, because in this situation the meaning is implied by the immediate context. Moreover, misunderstanding a local variable's use is expected not to have consequences beyond the surrounding block of code. Indeed, Kernigham and Pike write that "shorter names suffice for local variables; within a function, `n` may be sufficient, `npoints` is fine, and `numberOfPoints` is overkill" [14]. They then continue by giving an example where a loop is much clearer when abbreviated names are used, and conclude "Programmers are often encouraged to use long variable names regardless of context. This is a mistake: clarity is often achieved through brevity."

Studying the effect of variable name length is confounded by the question of what lengths to compare. We therefore focus on the extreme brevity of single-letter names: if these are found to be usable, less extreme abbreviations are probably usable too. We perform a sequence of three studies. The first uses repository mining to assess how commonly single-letter variables appear in the code of popular applications. As expected, `i` is the most common, but many other single-letter variables are also used. Interestingly, this is language dependent.

The second study is a controlled experiment in which students were asked to comprehend and modify a pair of functions in one of several versions: one in which variables are given full names, and others in which some variables are give single-letter names. The versions differed by the class of variables that were given the shorter names: they could be either local variables of limited scope or more substantial variables representing data structures. The results did not show any significant negative effect due to having some single-letter variables.

The third study considers possible meanings attributed to single-letter variables. This used a web-based survey in which respondents were requested to share associations elicited by different letters of the alphabet, and what variable types they would consider naming with each letter. We found that some letters are overwhelmingly associated with one or perhaps two types, and sometimes also with one dominant meaning.

Taken together, these studies make two main contributions. The first is to inform practice based on scientific evidence:

[1] https://google.github.io/styleguide/cppguide.html#General_Naming_Rules
[2] https://www.gnu.org/prep/standards/standards.html#Names

single letters are in fact used widely, they do not necessarily impair comprehension, and they may convey meaning. The second is to identify important knowledge gaps: for example, we show that certain letters have well-defined expected meanings, but this needs to be qualified by programming task, developer experience, and domain.

II. Related Work

While variable names are obviously important, and it has repeatedly been suggested that long and descriptive names be used, there has not been a lot of empirical research on the effect of variable names and in particular the possible drawbacks of single-letter names.

Coding standards such as those quoted above typically focus on stylistic aspects of variable names, such as the use of capitalization. Caprile and Tonella are among the few who propose explicit methodologies to come up with good names. Specifically, they suggest standardization of identifier names using a lexicon of concepts and syntactic rules for arranging them [7]. Likewise, Binkley et al. suggest rules for constructing field names by concatenating words representing different parts of speech [2]. These recommendations may be expected to lead to longish names, and they do not consider abbreviations.

On a more conceptual level, Deißenböck and Pizka develop a formal model of the mapping from concepts to names [8]. The model requires naming to be consistent (a one-to-one relationship of names to concepts) and concise (names should reflect the correct level of abstraction, and be neither too general nor too specific). They go on to suggest tools to aid in identifying naming issues, mainly synonyms as may be created by different levels of abbreviation.

One of the best-known studies of variable names is "What's in a Name?" by Lawrie et al. [15]. This introduced a controlled experiment similar to the one we performed, in which three versions of each of twelve functions are used: one with full names, one with abbreviations, and one with single letter names. Subjects were required to understand the code, and the results indicated that longer names led to better comprehension. However, in only 3 of the 12 code examples was the difference statistically significant. Our experiment differs by requiring subjects to also modify the code, and by avoiding the use of "iconic" functions like binary sort. Another similar experiment was conducted recently by Hofmeister et al. [11], and found that code with full-word names was faster to comprehend by 19% on average. But note that both the Lawrie study and the Hofmeister study were unmindful of the variables' use, and abbreviated *all* the variables. Our focus is on *identifying the cases* where single letter names can be used.

The possible detrimental effects of inappropriate naming were considered in a number of studies. Scanniello and Risi performed a controlled experiment in which students were required to find and fix faults in either of two versions of a program: one with full names and the other with abbreviated names [18]. The result was that the abbreviations did not hurt performance. We take the additional step of considering the option of single-letter names.

Butler and colleagues performed a number of studies on identifier names and naming conventions. In one they found that both long names (above 25 letters) and short ones (less than 8 letters, excluding some specific cases of commonly used single-letter and short names) are associated with both the cyclomatic complexity of methods and their Oman maintainability index [6]. In another, focused on adherence to naming conventions, they found that single-letter names (which they call "ciphers") are sometimes used as field names and not only as locals and formal arguments [5].

Interestingly, a number of studies have shown that *long* names may be implicated in reduced code quality. Binkley et al. show that long names tax programmer memory, leading to longer processing times and reduced recall [3]. Kawamoto and Mizuno studied the possible use of identifier name length in predicting faulty modules [13]. Their results indicate that longer names (not shorter names) are a predictor of faults. Aman et al. have shown that long local variable names are a predictor of change-proneness after release, implying lower quality [1]. They explain this effect by suggesting that long names may be "deodorant" used by programmers to compensate for problematic code. Similarly, in the readability model developed by Buse and Weimer, maximal identifier length provided a medium-level predictive input regarding a code snippet's readability [4]. But the minimal length and specifically the use of abbreviations and single-letter names were not checked.

Finally, an interesting historical analysis has been performed by Holzmann, showing how variable names have tended to become longer with time [12]. Binkley et al. suggest that they have reached their useful limit, as longer names become harder to remember [3].

III. Research Questions

Based on experience, Martin writes in *Clean Code* that "single-letter names can ONLY be used as local variables inside short methods. The length of a name should correspond to the size of its scope" [16]. We want to put such recommendations to an empirical test, by answering the following three research questions:

RQ1) To what degree are single-letter variable names used in practice?
- What fraction of variables are given single-letter names?
- Is such usage universal, or is it project or language dependent?

RQ2) Are single-letter variable names really harmful for program comprehension?
- What is the impact of changing variable names to single letters?
- Do all variables have equal impact on program understandability? More specifically, is it reasonable to use single letters for index variables and local variables?

RQ3) What do programmers expect when they see a certain single-letter variable name?

Together, these questions facilitate the assessment of how single-letter variables *are* used, and how they *may* be used, with implications to programming practice and education.

In answering the above questions, we use different experimental methodologies. The first question concerns the current state of the practice. Our answer is based on mining the github repository, and extracting data on the usage of single-letter variable names from the source code of multiple high-profile projects. The second question concerns the potential impact of single-letter names. To answer it we conduct a controlled experiment, where subjects are confronted with different versions of the same functions. The differences are that some variable names are replaced with single-letter alternatives. Finally, the third question relates to possible considerations when using single-letter variable names. We conducted an online survey to find what meanings programmers associate with the letters of the alphabet. Variables that are related to commonly found meanings can be expected not to cause problems if named with the appropriate letter.

IV. Usage Patterns of Single-Letter Variables

To answer the first research question we need to analyze the usage of different variable names by real programmers, spanning multiple programming languages and projects. Note that different programming languages and paradigms bring with them different coding conventions and idioms. These, in turn, may affect the style and structure of the resulting code. One way in which code bases differ from one another is in their use of different variable names: some use longer names while others prefer shorter names. In this research we specifically focus on the usage of single-letter names.

A. The Dataset

In order to collect the data needed for our research, we first needed to obtain a representative dataset of code to analyze. We decided to use github[3] as the source for the code, due to three important factors: it houses a large collection of recent projects, it allows easy categorization of each project by programming language, and it supports a ranking of project by their popularity (e.g., most starred, which probably reflects some notion of quality and actual use).

Given the diversity of programming languages and styles, a biased sample may lead to results that are representative of only the analyzed projects. To avoid this danger we constructed our dataset by first picking five programming languages with different characteristics, and then collecting the 200 most popular projects for each of these languages. The languages we picked are C, Java, JavaScript, PHP, and Perl, and represent conventional standalone programming, web programming in browsers and servers, and scripting. This procedure resulted in a large dataset (over 16 GB of source code), which is expected to represent each of the programming languages since it contains a large number of the most popular projects for each language.

B. Data Extraction

The projects themselves were collected by writing a small python script which automatically downloads the aforementioned projects from github.

[3]https://www.github.com.

In order to extract the variable names from each project we created a framework which enables the loading of plugins, dubbed "extractors", for different languages. These extractors know the language syntax and use this knowledge to retrieve variable name declarations from projects written in this language. The data is then stored in a local database. In order to allow for easy scaling and handling of large amounts of data, we elected to use MongoDB — a document database which excels in storing and searching over large amounts of data.

The technical details of the extractors for the different languages are as follows:

- **C** We extracted the abstract syntax tree (AST) from the `clang` compiler using a "syntax-only" execution (i.e., not requiring compilation, just performing lexing and parsing of the source file), and from the AST we extracted the variable names and types.
- **Java** We used an open-source project which enables easy parsing of Java source files[4]. We then wrote a plugin using this framework which allows extraction of variable names and types.
- **JS** We wrote a short script which uses a Node.js framework which analyses JavaScript source files[5] in order to dump the variable names. A special concern was "minified" JavaScript source files. As JavaScript files are often transmitted on the Internet together with web pages, keeping them as short as possible has operational advantages. Minification is a source-to-source transformation that reduces size by removing unnecessary white space and comments, and abbreviating variable names. We attempt to avoid minified JavaScript source files by excluding files with very few very long lines (indicative of minified code).
- **PHP** We used a project which parses PHP source code[6] in order to extract the variable names. As there are no variable declarations, we count definitions (that is, assignments to the variable). Note that variables in PHP are prefixed with a $, so technically single-letter variables are actually written with two letters.
- **Perl** We used the actual Perl syntax tree using a command-line flag in Perl itself (MO=Concise), and extracted the variable names from that data. Like PHP, Perl too prefixes variables with a $, which we ignore. Note that there are also many predefined special variables named using punctuation marks: `$_`, `$.`, `$,`, `$#`, and more. This increases the number of single-letter variables that are found.

C. Results

After collecting the dataset, we first looked at the distribution of variable name lengths in the selected projects in each language. We also compared the distribution for all 200 selected projects with the individual distributions of the 5 most popular projects, to verify that all look more or less the same and the distribution is not overly affected by some projects with unique characteristics.

[4]https://github.com/javaparser/javaparser

[5]http://esprima.org/doc/

[6]https://github.com/nikic/PHP-Parser

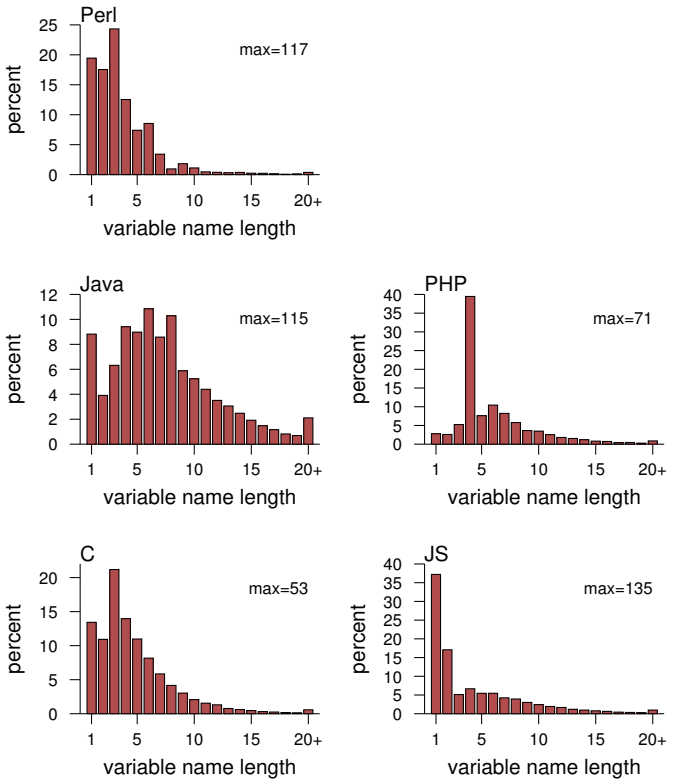

Fig. 1. *Distributions of variable name lengths in different programming languages.*

TABLE I. *Acceptable single letter variables in Java [10].*

b	byte
c	char
d	double
e	Exception
f	float
i, j, k	int
l	long
o	Object
s	String
v	arbitrary value

The results are shown in Figure 1. While single-letter variable names are typically not the most common, they are approximately as common as other short lengths except in PHP. In C, Java, and Perl they make up 9–20% of the names. Hence developers in many popular projects in diverse languages do not shy away from using single-letter variable names.

Next, we drew the histogram of single-letter variables in each language. This is shown in Figure 2. As may be expected, the most commonly occurring single letter variable name is `i`. This is most probably due to its use as a loop index. In some cases (notably C and Java) `j` is also highly used, probably for the same reason. But apart from that, the distribution is language dependent. The name `v` is highly used in Perl, even more than `i`. In C common names include `p` (which probably indicates a pointer), `c` (for chars), and `n` (presumably for counters). The Java Language Specification includes a section about naming conventions. This includes the acceptance of using single letter names to represent local

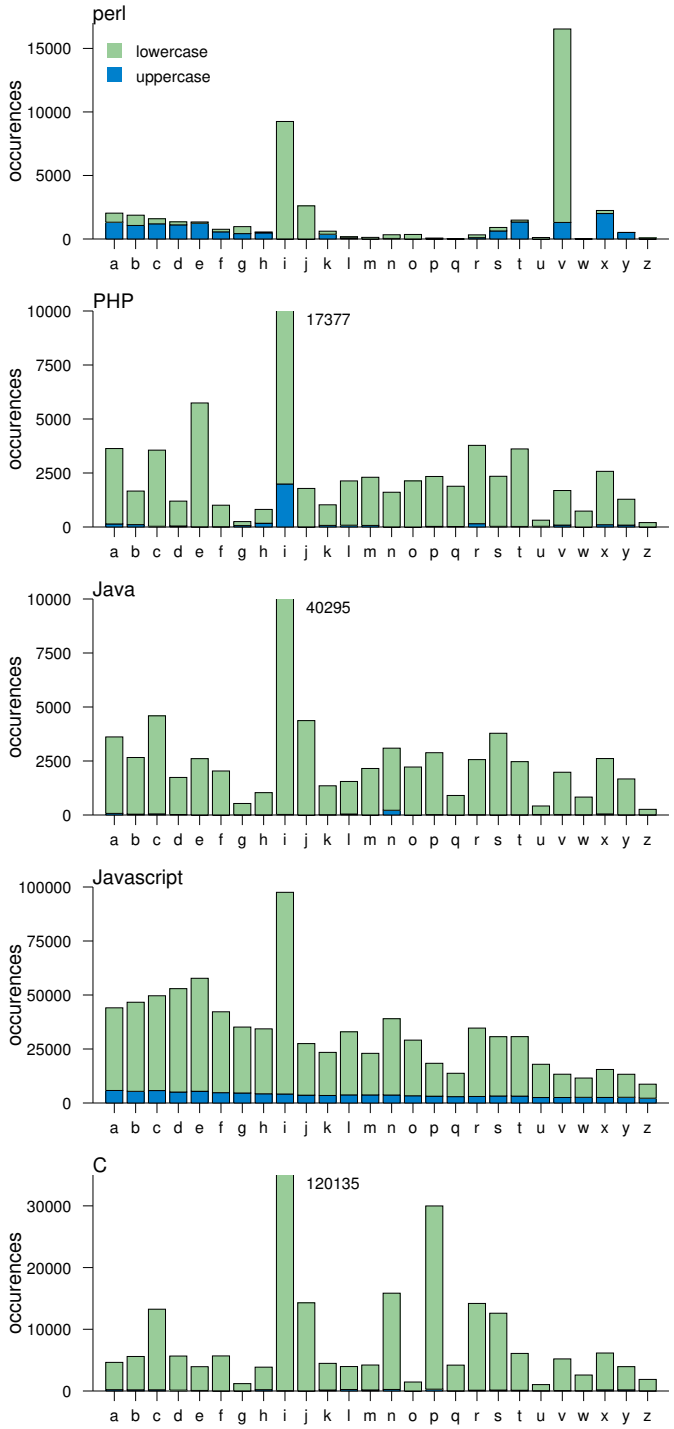

Fig. 2. *Histograms of single letter names usage in different programming languages.*

temporary or looping variables of different types, as listed in Table I [10]. Our results show that some but not all of these are indeed used more than other letters.

In addition, we observe that lowercase letters are used much more often than uppercase ones. Indeed, uppercase names are used mainly in Perl, where they typically outnumber

lowercase ones except for `i`, `j`, and `v`. In JavaScript uppercase names are more or less evenly distributed across all letters, and the variability of lowercase names is also somewhat smaller than for other languages. This (together with the fact that single-letter variables are very common as shown in Figure 1) may indicate that the dataset includes some minified code that was not identified correctly and removed by our filter.

V. Effect of Single-Letter Variables

To answer the second research question we need to assess the effect of single-letter variable names on programmer performance. We do this using three experimental procedures. The first two are controlled experiments where subjects are required to modify a given function. Different subjects receive different versions of the functions, either with full variable names or with single letter names. The third part is an opinion survey in which subjects indicate which version of a function they prefer. The same subjects performed all three experimental procedures in sequence.

A. Experimental Design and Execution

The experiment consisted of a sequence of two controlled experiments and a survey. We used two functions in the controlled experiment setting to reduce the threat to validity arising from having only a single data point. The functions were chosen based on three criteria: they are real code (not concocted examples created for the experiment), they are concise and well-defined, and they do not require specific domain knowledge. One function was written in Java and the other in C. In the first, three versions were used with different patterns of changed variable names as described below. In the second two versions were used.

The controlled experiments were executed by providing the functions on paper. The assignment of experimental treatments for each function was randomized. In total there were therefore six combinations of version of the first function and version of the second function. The survey was administered via computer.

In performing the experiment subjects went through the following steps:

1) Sign a consent form and be paid up front.
2) Fill in the demographic details questionnaire.
3) Read instructions and perform first experiment.
4) Read instructions and perform second experiment. This experiment had two discrete steps.
5) Read instructions and perform third experiment.

Overall this typically took 30-45 minutes in total, although some stayed longer. Subjects were allowed to leave at any step of the experiment, and several did so without completing the second experiment (which turned out to be the hardest). No identifying information was collected.

B. Experimental Subjects

The experimental subjects were Hebrew University students in the computer science and computer engineering programs. Some had a double-major with some other program, such as mathematics or cognitive sciences. Subjects were recruited from the student programming labs where they spend time on self study and solving programming exercises in their various courses. They were paid 50 NIS (approximately $13) for participation. We found that physically walking through the lab and inviting potential subjects to the experiment worked much better than distributing fliers and waiting for subjects to contact us. It also saved the need to schedule experimental sessions and coordinate with multiple subjects.

In total 56 students were recruited. 41 were male, 13 female, and 2 did not report their sex. The vast majority were undergraduates, with 31 completing their first year (2 semester-long programming courses), 9 completing the second year, and 12 the third year. 2 were MSc students. The average age was 23.9 years.

C. Experiment 1

Our goal is to characterize the effect that single letter variable names have on code clarity and understanding. In this experiment we attempt to measure the negative effect of single-letter variables on code, in combination with the question of whether it is justified to exclude local and index variables from the requirement of meaningful variable names.

1) Experimental Treatments: The first experiment was based on a function that receives a sequence of URLs and completes the sequence. The scenario is a web crawler collecting data from an online forum. Forum pages usually have links to other pages at the bottom, to enable non-sequential browsing, but if the forum is long the list may include only the first few pages and the last one. The function fills the gap in order to enable parallel retrieval of all the pages. This is based on identifying a common pattern (which is given) and completing a sequence of serial numbers. However, it has a deficiency that if the list comes from the first page then number 1 will be missing, because links are given only to *other* pages. The experimental task is to fix this defect. Subjects are given a full explanation with examples at the outset. This is meant to differentiate between comprehending the *functionality* of the code and understanding the *implementation* of this functionality. We want participants to focus on understanding the code itself to find where to make the fix.

In terms of code, the function accepts two parameters and is a full page long, including inline documentation. It has 9 local variables, of which 2 are in a loop, and 3 are lists. There is no documentation about the variables. Given this function, we created three versions with different characteristics:

V1) Full identifier names
V2) Both loop identifiers and a related local given single-letter names
V3) Both parameters and the most important list (the page numbers) given single-letter names

This was meant to enable two types of observation: first, whether single-letter variables have an impact on comprehension, and second, whether it matters *which* variables are those that are given single letter names. Specifically, it may be acceptable to give index variables and other variables of local and limited scope a single-letter name, but perhaps comprehension is facilitated when parameters and major data structures are given meaningful names.

TABLE II. *Results of experiment 1: time to solution of successful subjects and rate for all subjects. ± denotes standard deviation.*

ver	succ	time [m]
V1	11	15.5±4.4
V2	13	18.6±6.3
V3	18	17.4±5.9

ver	N	rate
V1	17	.044±.036
V2	18	.042±.029
V3	21	.054±.029

2) Results and Analysis: The function works by identifying common substrings in the given URLs, extracting page numbers (which obviously are not common to different URLs), completing the set of page numbers, and then generating URLs for the complete set. The best solution is therefore adding code to add the integer 1 at the beginning of the already-completed list of page numbers, in case it does not already start with 1. We coded the solutions on a 4-point scale as follows:

0 no solution or wrong
1 inferior, e.g. generating the URL for 1 manually and appending at the end
2 correct idea but with minor issues
3 perfect as described above

In addition we measured the time to solution.

To compare the times it took subjects to cope with the different versions, we employ the following null hypothesis:

> H_0: The time required to cope with version A and version B is the same.

This is applied to all three pairs of versions. We use SPSS for the statistical analysis, first applying Levene's test for equality of variances, and then either the t-test or Welch's test to see if the null hypothesis should be rejected at $\alpha = 0.05$ (the t-test can be used only if variances are equal).

Comparing the times of all subjects leads to the result that the null hypothesis cannot be rejected, but this is uninteresting, because it mixes the times of those who solved the problem correctly with the times of those who failed or gave up. We therefore need to take the scores into account when comparing times. We do so in two alternative ways. The first is to compare the times of only those subjects who succeeded, and received scores of 2 or 3. However, this scheme loses data about how many subjects received lower scores. The second is to binarize the score, with 0 representing failure or an inferior solution (scores of 0 or 1) and 1 representing success (scores of 2 or 3). Inferior solutions were grouped with failed solutions rather that with good solutions because they reflect trying to circumvent the code rather than understanding it. We then calculate the success *rate* as the binarized score divided by the time. This includes the data about subjects who did not do well and effectively creates a continuous transition from taking a very long time to failing.

The results are shown in Table II. It is immediately obvious that the averages are close to each other, and the differences are smaller than the standard deviations. And indeed, the t-tests use to compare pairs of versions to each other all indicated that the null hypothesis could not be rejected. None of the p-values were even close to 0.05. Note, however, that this can also be the result of insufficient statistical power. The power, in turn, depends on the effect size. As our results indicate that the effect size is probably small, a much larger sample may be needed to bring differences to light. However, if the effect size is small then these differences may not be very meaningful even if they exist. In any case our present results should be interpreted as failing to show a difference, and not as finding that there is no difference.

TABLE III. *Results of experiment 1: odds ratio for the two extreme versions.*

ver	fail	success	odds
V1	6	11	1.833
V3	3	18	6.000

To compare scores we calculate the odds of success (score of 2 or 3 relative to score of 0 or 1) for each version, and compare versions with an odds ratio test. Odds ratios are used to see how an effect changes the odds for an outcome. A common example is the effect of smoking on cancer. We have four groups of subjects, with all combinations of smoking or not and having cancer or not. The question is how the odds of having cancer depend on smoking. This is quantified by how the odds for smokers differ from the odds for non-smokers. In our case, the question is how the odds of receiving a high score differ for subjects faced with one version of the program or another version.

The raw results are shown in Table III, with the calculated odds to succeed. The odds ratio is then found to be 3.272, meaning that the odds for success are more then 3 times higher in version V3 (which has single letter names for important variables). However, the 95% confidence interval spans the range [0.68, 15.8], which includes 1. Therefore again we cannot reject the hypothesis that the odds are actually the same.

It should be noted that our results can be seen as similar to those of Lawrie et al. [15], who also compared programmers trying to understand codes with full-name variable, abbreviations, and single letters. While they did find statistically significant differences in 3 cases, their differences were not significant (like ours) in another 9 cases.

To better understand the effects of all variables we used ANOVA with time or rate as the dependent variable, and version and demographic variables as the explanatory variables. The resulting models were poor in the sense that they explained only a small percentage of the variance (7–11%), and were not statistically significant. Even so, they indicated that the program version (treatment) had the smallest effect, and that age and sex had a much larger effect.

D. Experiment 2

This experiment is focused on the possible adverse effects of single-letter names, using a more extreme treatment than the previous one.

1) Experimental Treatments and Tasks: This experiment uses a very short function written in C, whose aim is to count the number of set bits in a given array of bytes. It uses a predefined lookup table to count the number of set bits in each nibble of the given bytes. The table would look like a random series of numbers to the layperson. We have created two versions of this function: one containing informative variable names, the other containing only single-letter variable names, as shown in Figure 3.

Full names:

```
uint32_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] & 0xF] +
                 num_set_bits[(data[i] >> 4) & 0x0F];
    return count;
}
```

Single-letter names:

```
uint32_t n[] = { 0, 1, 1, 2,
                 1, 2, 2, 3,
                 1, 2, 2, 3,
                 2, 3, 3, 4 };
uint32_t f(uint8_t* d, int l) {
    uint32_t c = 0;
    for (int i=0; i<l; i++)
        c += n[d[i] & 0xF] +
             n[(d[i] >> 4) & 0x0F];
    return c;
}
```

Fig. 3. *Versions of the C function used in experiment 2.*

TABLE IV. *Results of experiment 2: time to solution of successful subjects.*

task	ver	N	succ	time [m]
1	V1	26	7	16.9±8.5
	V2	28	8	15.2±7.7
2	V1	22	7	10.0±5.0
	V2	24	6	9.7±4.4

Subjects are required to perform two experimental tasks. The first is to deduce the purpose of the function. Once done, we reveal the actual specification of the function. Subjects then proceed to the second part, where they are asked to extend the function to count pairs of bits, instead of single bits. This allows us to check whether variable names have any effect on the depth of understanding of the code. For example, a shallow level of understanding might be more prone to making mistakes when attempting to modify the code.

2) Results and Analysis: As in the previous experiment, times were recorded separately for both parts and both were separately graded on a scale of 0 to 3. We then performed exactly the same statistical analyses described above on both parts. Results are shown in Table IV.

An important observation is that relatively many subjects failed to understand the function in the first part, and even after it was explained at the outset of the second part, many failed to modify it. Thus this function was probably too hard relative to the programming knowledge of at least some of the student subjects.

As in experiment 1, the results indicated a lack of statistically significant differences between the experimental treatments — both groups (with or without meaningful identifier names) succeeded (or failed) to a similar degree. This implies that factors such as individual differences, domain knowledge, or technical knowledge have a much more significant impact than variable names: if you do not have the required background and skills, good variable names will not save you.

E. Experiment 3

In order to assess how participants feel about functions with single-letter variable names compared to meaningful variable names, we presented each subject with four questions, all of the type "which function do you prefer", intentionally not specifying the characteristics by which they should make a choice. Three questions included all pairings of 3 versions of one short function, and the fourth compared two versions of another longer function. The short function shuffles the elements of an array, and the versions are similar to those used in experiment 1 above: all long names, only local and index variables given single-letter names, and all variables given single-letter names. The long function does a deep comparison of two objects. The difference between the versions is calling them `a` and `b` or rather `first` and `second` (and likewise for related variables, such as `aStack` vs. `firstStack`). Both functions are real life code from the Underscore.js library.

The survey results were that in each comparison between 71% and 82% preferred the longer version. The biggest difference occurred when comparing full names to single letter names. Thus it can be clearly concluded that an overwhelming majority prefers methods with longer, more meaningful names.

Interestingly, this result differs from the results of the two controlled experiments reported previously. Those experiments indicate that there is no significant difference in how programmers cope with modifying a function when variables have full names or single letter names. In particular, we found that the variable names have a much smaller effect than other independent variables, especially demographic ones. But despite this lack of actual effect, the programmers reported that they prefer the longer more meaningful names.

VI. Associations of Single-Letter Variables

As noted above, it is pretty common to use `i` as a loop index, so when programmers see `i` they may expect it to be a loop index. But what about other letters, such as `s` or `t`? Are they also associated with a common meaning that can help comprehension? To answer this question we conducted an online survey. In this survey we ask about the associations with all letters of the alphabet.

A. Survey Structure

The survey starts with basic demographic details about each subject. The collected details are age, gender, education level, years of experience in programming, number of programming languages, and different programming level skills.

The survey aims to question the subjects about each single letter in the English alphabet. This means there are 26 questions. Each such question has two parts. In the first we ask for what types would you consider naming a variable by this letter. The possible answers are:

- Integer

- String
- Char
- Array
- Boolean
- Float/Double
- User defined type
- Other (free text answer)

The second part is the open question "What associations spring to mind when you see a variable named 'a'?" (of course using a different letter in each question).

Because the English alphabet has 26 letters our survey is very long, with 26 questions that each has two parts. In addition the questions are very repetitive, as they all have exactly the same structure. There is therefore a danger that subjects will lose interest and abandon the survey in the middle, or just pause it and fail to return. As a result we will have much less data for letters toward the end of the alphabet. To mitigate this danger we need to randomize the order that the letters are presented. Thus some subjects will get the question about `a` at the beginning of the survey, but others will get it in the middle or the end. And on average we will get enough data about all the letters even if subjects quit during the survey.

B. Survey Platform and Administration

Very many platforms for conducting online surveys are available, and we checked 14 of them. Our considerations for selecting a platform were the following:

- Support for dividing the survey into pages. We need a separate page for each letter, so we can randomize the order.
- Support for randomization of the questions.
- Saving partial results in case the user abandons the survey in the middle.

In addition, we naturally prefer low-cost (or free) platforms. In the end we selected Qualtrics[7] as our platform, as it supports all our requirements and allows a single survey for free.

In order to attract subjects we used two methods: word of mouth among friends and colleagues, and advertising in online forums. We mainly focus on reddit as a major source, and posted the survey on three different channels: SampleSize, Programming, and AcademiCode.

C. Results

Altogether 96 subjects entered the survey. 35 of them completed all the questions, and 15 left without answering even a single question. 62% of the participants were male, 17% were female, and 21% did not specify their sex. Ages ranged from 16 to over 60, with the majority between 17 and 32. Respondents seem to be quite experienced, with 30% claiming 10 years of experience or more, and 23% claiming knowledge of 6 or more programming languages. The most commonly known languages were Java and JavaScript. 33 of them had a Bachelor's degree, 19 a Masters, and 3 a Doctorate. An additional 12 had some college training but no degree.

[7]https://www.qualtrics.com/

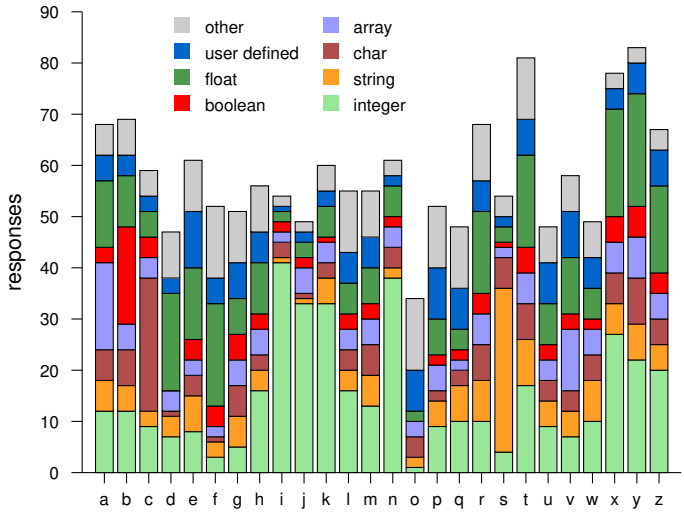

Fig. 4. *Considered types for each letter of the English alphabet.*

The types that would be considered for variables names with the different letters of the alphabet are shown in Figure 4. Note that some respondents noted several optional types in some cases, so the total may be larger than the number of respondents. As may be expected, in some cases the type that is associated with a letter starts with this letter. Examples are `s` for string and `c` for char. But in other cases this is not so dominant, as in the case of `b` and boolean or `a` and arrays. Likewise, `o` tends to be associated with object (as noted under "other").

In other cases the type does not start with the letter. For example, `i`, `j`, `k`, and `n` are all strongly associated with integers. Somewhat surprisingly, `d`, `e`, `f`, `r`, and `t` tend to be associated with floating point. And interestingly, the generic variables `x`, `y`, and `z` are associated with integers and floating point to a similar degree.

The common interpretations for each letter are listed in Table V, and statistics of the results are shown in Figure 5. Each letter received around 30 answers on average (bottom panel). Interestingly, many participants bothered to note explicitly when they had no associations (orange overlay), and this differed considerably from letter to letter: nearly everyone had associations for `i` and `j`, but half or more didn't for `h`, `q`, and `u`. In many cases the associated meanings were very diverse, with up to 21 different meanings suggested for the same letter (middle panel). However, in some cases many of the responses were in fact concentrated in a single meaning (top panel, excluding no-association replies).

High concentrations include the letters `s`, which very many took to mean "string", and `t` which was associated with "time". `i`, `j`, and `k` were all associated with "loop index" (or just "index"). Interestingly, the highest concentration occurred for `j` and not for `i`. That happened because `i` had more competing interpretations, including "integer", "counter", and "temporary variable". The same effect occurred with the meaning "coordinate", which was the most common interpretation for `x`, `y`, and `z`. But `x` had some other common meanings, so the concentration on "coordinate" was lower.

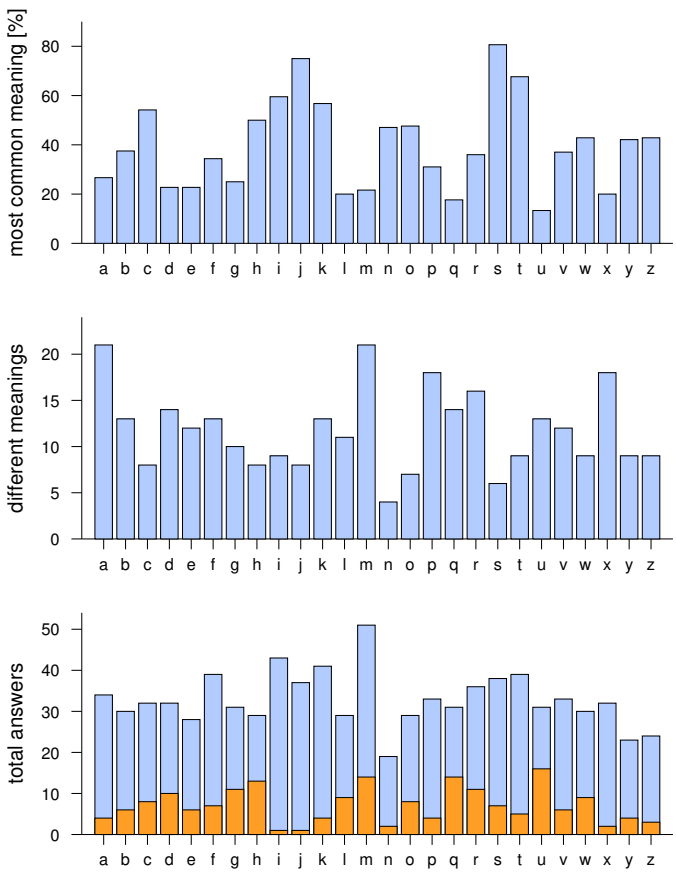

Fig. 5. *Number of interpretations given to variable named by different letters of the alphabet. Overlay in bottom panel is responses of "none".*

TABLE V. *Common meanings associated with different letters of the alphabet. (Number in parentheses is number of occurrences.)* ***Boldface*** *entries were suggested by 50% or more of total participants who answered; italics denotes letters where the maximum was supported by 20% or less.*

a	array(8) counter(3) temporary(3)
b	boolean(9)
c	character(13) counter(5)
d	*double(5) distance(3) date(3)*
e	*math e(5) error(3) event(3) exception(3)*
f	float(11) function(10)
g	*global state(5) function(4) local const(3)*
h	height(8)
i	**loop index(19) index(6)** integer(5) counter(4) temporary(3)
j	**loop index(21) index(6)** counter(3)
k	**loop index(17) index(4)** constant(3)
l	*length(4) long(3) loop index(3)*
m	*counter(8) number(5) member(3)*
n	counter(8)
o	object(10) zero(4)
p	pointer(9) probability(3)
q	*query(3)*
r	radius(9)
s	**string(25)**
t	**time(23)** temporary(4)
u	*[no meaning]*
v	vector(10) matrix(3)
w	weight(9) wide(5)
x	*coordinate(6) math operation(5) temporary(3)*
y	coordinate(8) math example(3)
z	coordinate(9) data(4)

Another interesting effect is the discord between the associated meanings and the actual usage. Table V shows that `i`, `j`, and `k` were all associated with indexes to similar degrees. But Figure 2 shows that `i` is very heavily used, `j` is used somewhat, and `k` seldom.

Considering other letters, opinions about `f` were nearly equally divided between "float" (floating point variable) and "function". There are also some ambiguities. For example, the results above for `i`, `j`, and `k` actually include both "loop index" and just "index"; for `i` a couple of subjects also suggested "array index". For `w` a relatively common interpretation was "wide", which is related both to "wide string" (meaning Unicode as opposed to ASCII) and to "width" which were mentioned only once each.

VII. Threats to Validity

The threats to validity are different for the different studies reported. In the initial repository mining, an external threat is that our results exhibit a variability between languages. They therefore may not pertain to projects written in other languages. But the most salient results (that single-letter variable names are in fact used, and that `i` is very common) are probably universal. Moreover, for each language studied we used 200 popular projects from github; other projects may exhibit different patterns, although an individual comparison with the top 5 projects showed the overall results to be representative. Finally, in the case of JavaScript we may have inadvertently included some minified code. This is a potential construct validity issue.

In the controlled experiment study, a common problem is variability among experimental subjects which leads to a threat to internal validity. We could not use a within-subject design to control for this, due to the expected learning effect: once a subject sees a function in one treatment, you cannot use the same function in an alternative treatment. Also, other effects such as stress and fatigue may be at play. However, we did mitigate all these concerns, at least to some degree, by random assignment of experimental treatments.

Another commonly cited issue is the use of students as experimental subject. We think students are adequate for basic programming tasks like the ones we used [9]. However, we may have had too many first year students, as witnessed by their difficulties in solving the second task. This resulted from their lack of schooling in specific required technologies, mainly bit operations in C, which forced us to give explanations during the experiment. Students may also be problematic for opinion tasks, such as preference for a coding style, as they are influenced by the requirements of TAs in an academic setting ("you will lose points if you use cryptic identifier names") and this is not offset by real-world experience.

An additional threat is that only two functions were used, and these functions may not be the most revealing for this type of issue, i.e. maybe the variable names are not the most important thing in these specific functions. Also the treatments may not have been extreme enough, as we changed only some variables and not all of them. Our results therefore cannot be generalized to a claim that single letter variables do not have any effect in general.

The survey regarding associations and types suffers from the threats associated with any online survey, namely lack of control over participants. We do not really know who they are and how serious they are. In addition fatigue effects may be present, but these are mitigated by randomizing the order of the questions.

VIII. Conclusions

Single letter variable names are used in many projects across programming languages, and may constitute 10–20% of all names. This reflects the advice given by Kernighan and Pike that "Local variables used in conventional ways can have very short names. The use of `i` and `j` for loop indices, `p` and `q` for pointers, and `s` and `t` for strings is so frequent that there is little profit and perhaps some loss in longer names" [14].

However, which letters are widely used is language specific, and probably reflects the domains in which the language is used. For example, the use of `p` for pointer variables is probably confined to C. Furthermore, their conjecture that `q` and `t` would also be frequently used for pointers and strings, respectively, is not supported by the data collected (because they are used much less frequently than `p` and `s`). This underlines the need to collect and analyze real data.

The main motivation for out work was to contribute to the discussion of whether and when single letter variable names may be used. In our experiments we did not find any significant detrimental effect of such variables on comprehension. This implies that programmers can cope with such variables, and can use them without significant ill-effects. However, more work is needed to complement these results using additional functions and treatments, e.g. with few or many variable name substitutions. Using more complicated functions with numerous variables may identify situations in which single letter variable names do have a deleterious effect.

At the same time, we note that our work does not cover all aspects of variable naming. For example, Martin notes that it is important for variable names to be searchable [16], and single letter names fail this test. Likewise certain letters should most probably be avoided, like lowercase L and uppercase O, which look like 1 and 0.

As a side effect, the results indicate that domain knowledge and experience with specific technologies may trump the effect of variable names. In addition, individual differences between experimental subjects have a very strong effect. So another interpretation of the results is that variable names are simply not that important relative to other factors.

Finally, we found that specific letters (in addition to `i`) are in fact strongly associated with certain types and meanings. Using these letters in these specific contexts is therefore expected to be benign and safe, and provides a possible mechanism that explains why use of single letter names had little if any detrimental effect. Indeed, our results should be interpreted as suggesting that *certain single letter variable names can be used in some roles*; this does not imply that *all* single-letter variable names are *always* acceptable.

Further empirical work is required to elucidate the details of how single letter variable names interact with the specific letter and context. A good starting point would be to extend our initial survey of the associations between letters, types, and meanings. This can be combined with a more detailed study of the roles that single letter variables play in existing code (e.g. what is `v` used for in Perl?). Also, an open ended survey of programmers would allow them to explain when they use single letter variables. Sorting the answers into categories would then enable the derivation of a taxonomy of uses.

Verifiability

In the interest of verifiability and reproducibility, our experimental materials and results are made available at `https://bitbucket.org/sophiko/single-letter-variables-icpc-2017`.

Acknowledgments

This research was supported by the ISRAEL SCIENCE FOUNDATION (grant no. 407/13).

References

[1] H. Aman, S. Amasaki, T. Sasaki, and M. Kawahara, "*Empirical analysis of change-proneness in methods having local variables with long names and comments*". In *Intl. Symp. Empirical Softw. Eng. & Measurement*, pp. 50–53, Oct 2015, DOI: 10.1109/ESEM.2015.7321197.

[2] D. Binkley, M. Hearn, and D. Lawrie, "*Improving identifier informativeness using part of speech information*". In 8th *Working Conf. Mining Softw. Repositories*, pp. 203–206, May 2011, DOI: 10.1145/1985441.1985471.

[3] D. Binkley, D. Lawrie, S. Maex, and C. Morrell, "*Identifier length and limited programmer memory*". *Sci. Comput. Programming* **74(7)**, pp. 430–445, May 2009, DOI: 10.1016/j.scico.2009.02.006.

[4] 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.

[5] S. Butler, M. Wermelinger, and Y. Yu, "*Investigating naming convention adherence in Java references*". In 31st *Intl. Conf. Softw. Maint. & Evol*, pp. 41–50, Sep 2015, DOI: 10.1109/ICSM.2015.7332450.

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

[7] B. Caprile and P. Tonella, "*Restructuring program identifier names*". In *Intl. Conf. Softw. Maintenance*, pp. 97–107, Oct 2000, DOI: 10.1109/ICSM.2000.883022.

[8] F. Deißenböck and M. Pizka, "*Concise and consistent naming*". In 13th *IEEE Intl. Workshop Program Comprehension*, pp. 97–106, May 2005, DOI: 10.1109/WPC.2005.14.

[9] 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].

[10] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, *The Java Language Specification: Java SE 8 Edition*. Oracle America, Inc., 2015.

[11] J. Hofmeister, J. Siegmund, and D. V. Holt, "*Shorter identifier names take longer to comprehend*". In 24th *IEEE Intl. Conf. Softw. Analysis, Evolution, & Reengineering*, Feb 2017.

[12] G. J. Holzmann, "*Code clarity*". *IEEE Softw.* **33(2)**, pp. 22–25, Mar/Apr 2016, DOI: 10.1109/MS.2016.44.

[13] K. Kawamoto and O. Mizuno, "*Predicting fault-prone modules using the length of identifiers*". In 4th *Workshop on Empirical Softw. Eng. in Practice*, pp. 30–34, Oct 2012, DOI: 10.1109/IWESEP.2012.15.

[14] B. W. Kernighan and R. Pike, *The Practice of Programming*. Addison-Wesley, 1999.

[15] 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.

[16] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftmanship*. Prentice Hall, 2009.

[17] S. McConnell, *Code Complete*. Microsoft Press, 2nd ed., 2004.

[18] G. Scanniello and M. Risi, "*Dealing with faults in source code: Abbreviated vs. full-word names*". In 29th *Intl. Conf. Softw. Maintenance*, pp. 190–199, Sep 2013, DOI: 10.1109/ICSM.2013.30.