Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Identifier length and limited programmer memory

Dave Binkley[*], Dawn Lawrie, Steve Maex, Christopher Morrell

Loyola College, Baltimore MD, 21210-2699, USA

ARTICLE INFO

Article history:
Received 2 September 2008
Received in revised form 15 February 2009
Accepted 18 February 2009
Available online 11 March 2009

Keywords:
Program comprehension
Memory
Identifier names

ABSTRACT

Because early variable mnemonics were limited to as few as six to eight characters, many early programmers abbreviated concepts in their variable names. The past thirty years have seen a steady increase in permitted name length and, slowly, an increase in the actual identifier length. However, in theory names can be too long for programmers to comprehend and manipulate effectively. Most obviously, in object-oriented programs, entity naming often involves chaining of method calls and field selectors (e.g., class.firstAssignment().name.trim()). While longer names bring the potential for better comprehension through more embedded sub-words, there are practical limits to their length given limited human memory resources.

The driving hypothesis behind the presented study is that names used in modern programs have reached this limit. Thus, a goal of the study is to better understand the balance between longer, more expressive names and limited programmer memory resources. Statistical models derived from an experiment involving 158 programmers of varying degrees of experience show that longer names extracted from production code take more time to process and reduce correctness in a simple recall activity. This has clear negative implications for any attempt to read, and hence comprehend or manipulate, the source code found in modern software. The experiment also evaluates the advantage of identifiers having probable ties to a programmer's persistent memory. Combined, these results reinforce past proposals advocating the use of limited, consistent, and regular vocabulary in identifier names. In particular, good naming limits individual name length and reduces the need for specialized vocabulary.

1. Introduction

Early compilers limited identifier length starting a long tradition of abbreviation [11]. Modern programming languages have forgone such limits; thus, programmers have been encouraged to replace abbreviations, such as mnelmns, with ever longer names, such as max_num_elmns or even max_number_elements. However, a name can become too long. Consider trying to work with the_maximum_number_of_elements_seen_in_the_input even given an IDE that performs name completion.

The effective removal of length limitations provides programmers with considerable freedom to select names that promote source code understanding. With this freedom comes responsibility. For example, Liblit et al. observe that longer names, with more embedded sub-words, are more informative [11]. Furthermore, Lawrie et al. note that longer identifier names provide insight into the code's meaning [9,10].

Yet, there is a practical limit to the length of identifiers, just as there is with the length of words in natural language. Jones observes that few developers appreciate how small short-term memory actually is, having the capacity to hold only

[*] Corresponding author. Tel.: +1 410 617 2881.
E-mail addresses: binkley@cs.loyola.edu (D. Binkley), lawrie@cs.loyola.edu (D. Lawrie), smaex@cs.loyola.edu (S. Maex), chm@loyola.edu (C. Morrell).

information on a few statements at most [7]. Thus, overloading a programmer's short-term memory is rather easy. One way to improve its capacity is by constructing identifiers from words that include ties to programmer persistent memory.

This paper presents results from a study of the interplay between programmer short-term memory limitations and entity names found within programs. The study is part of ongoing research into the impact that limited human memory resources have on program comprehension. While program comprehension clearly involves more than simple recall, such recall forms a fundamental and necessary precursor to essentially all comprehension activities. Thus, higher-level comprehension activities become increasingly difficult as a programmer's short-term memory becomes overcrowded, particularly when those activities require access to crowded-out information. It is important to separate this observation from the observation that low recall is a sign of poor comprehension. Only the former is investigated by the experiment.

In contrast to trends in modern programming languages, the hypothesis that motivates this study is that identifiers can be too long and thus negatively impact recall and consequently comprehension. While the existence of a theoretic limit is obvious, the study suggests that this limit has, in fact, been reached. The resulting degradation can be observed, for example, as a reduced ability to recall particular parts of the code caused by overcrowding of short-term memory. This is particularly an issue in object-oriented programming where 'chaining' often leads to long names created when several method calls and field selectors are concatenated together in a *chain expression* (*e.g.*, name.trim().length()). Informally, Jones notes this effect in the observation "experience shows that developers sometimes read source code so quickly that visually similar, but different, identifiers are treated as being the same identifier [7]".

Maximal comprehension occurs when the pressure to create longer more expressive names is balanced against limited programmer short-term memory resources. Finding this balance point will result in better guidance for creating easy-to-comprehend names. To this end, results from a study on the impact of name length and persistent-memory ties are presented. An important part of the study's construction is the use of production code as a source for the chain expressions (names). This means that the results are more apt to apply to existing rather than hypothetical programs. However, the study considers isolated names; therefore, it tends to underestimate the demand on programmer memory because there is no need to remember surrounding code.

A programmer's ability to quickly read (visually process an identifier's text) and to associate it with a concept forms a fundamental step in most program comprehension activities. Excessive length can get in the way of both of these processes. For example, compare the ease of comprehension between the statements cartesian_distance = square_root(distance_between_abscissae * distance_between_abscissae + distance_between_ordinates * distance_between_ordinates) and dist = sqrt(dx * dx + dy * dy). While program comprehension is more than simple reading and recall, these two steps are at the core of many program comprehension activities. This problem is exacerbated in source code because it includes an intensive use of invented words, unlike natural language in which participants share a large and mutually understood, but relatively fixed lexicon.

The remainder of the paper first presents some necessary background information in Section 2 before describing the experiment's design, the five hypotheses considered, and the results in Sections 3–5, respectively. This is followed by a discussion of related work in Section 6. Finally, Sections 7 and 8 suggest some future work and finally summarize the paper.

2. Background

This section describes background on memory, the findings from two prior motivating studies, and finally the statistical techniques used. *Memory* is the retention of information over time. It can be broken down into three stages: *sensory, short-term*, and *persistent*. *Sensory memory* retains images, sounds, and smells for no more than a second. Interesting information enters *short-term memory*. This second kind of memory is working storage. Short-term memory holds new information for about 15–30 s without rehearsal (repetition) [13]. The capacity of this subsystem is debatable, but limited. Miller first enumerated the capacity in the 1950s at seven plus or minus two *items*. However, more recent estimates indicate that it may be closer to three plus or minus one item [5]. One of the interesting aspects of short-term memory is that similar material can be combined into a *chunk*; this appears to increase the total amount of material remembered. Finally, *persistent memory* lasts from minutes to years. Persistent memory can be used to recall previously learned information including, for example, specific rules and repeated patterns such as those used in programming.

This study focuses on the impact of limited short-term memory resources. When dealing with natural language reading, short-term memory is correlated with reading comprehension. For example, Table 4.7 found in the Technical Manual for the Wechsler Memory Scale and Wechsler Adult Intelligence Scale [4] reports that the correlation between working (short-term) memory and reading comprehension as 0.65. The manual places this value in the high correlation range stating ". . . scores are high, usually in the 0.60 s and the 0.70 s . . ." [4]. It is interesting to note that the correlation between short-term memory and IQ is only 0.29, which is just below the moderately-correlated range of 0.30–0.60.

In the first of the two prior studies, programmer comprehension was measured using three different *levels* of identifiers: short (often single letter), abbreviated, and full word [9,10]. Unexpectedly, programmers elicited essentially the same information from full word and abbreviated identifiers. The second prior study used information from the English Lexicon Project. It showed that longer words required significantly greater time to determine their validity as actual words (greater lexical decision making time) [14]. One possible explanation for this phenomenon is that longer words require multiple fixations (landings of the eye) before they can be recognized and correctly classified.

Combined, the results of these two studies suggest a study of the impact of limited short-term memory on source code comprehension. In other words, can names of the lengths used in modern programs be *too* long and thus crowd short-term memory? The study described herein presents statistical models based on data collected to investigate this question.

As the statistical models are constructed from data that includes repeated measures and missing values (e.g., due to participant drop out) linear mixed-effects regression models were used to analyze the data [17]. Such models easily accommodate unbalanced data, and, consequently, are ideal for this analysis. These statistical models allow the identification and examination of important explanatory variables associated with a given response variable.

The construction of a linear mixed-effects regression model starts with a collection of explanatory variables and may include a number of interaction terms. The interaction terms allow the effects of one explanatory variable on the response to differ depending upon the value of another explanatory variable. For example, if Java Experience interacts with Sex in a model where Correctness is the response variable, then the effect on Correctness of Java Experience depends on Sex (i.e., is different for men and women). Backward elimination of statistically non-significant terms ($p > 0.05$) yields the final model. Note that some non-significant variables and interactions are retained to preserve a hierarchically well-formulated model [12]. Therefore, individual p-values for terms participating in an interaction are not reported. The quality of a model is reported using its *Akaike Information Criterion (AIC)* [17], which is based on the log-likelihood but is penalized for the number of parameters in the model. AIC can be used to compare mixed-effects models. The model with the smallest number is preferred.

When interpreting the mixed-effects models described in Section 5, graphs are used to illustrate significant effects in the model. However, when the models have more than two explanatory variables it is not pragmatic to graph all the variables. Thus, when plots are constructed, variables not being discussed are set to their means or, in the case of categorical variables, a representative value.

3. Experimental design

The explanation of the study's design begins with a description of the source code selected and then presents the layout of the experiment including the web-based applet used to collect the data, the explanatory and response variables collected, and the preparation of the raw data for analysis. The next section presents the five hypotheses formally studied using the collected data.

To investigate the interplay between memory and identifier selection, the study considers how well subjects retain the information contained in Java *chain expressions*. In doing so, it explores two orthogonal questions: what is the impact of the length of a chain expression on retention and what is the impact of familiarity with the parts (identifiers) that make up the chain expression on retention.

3.1. Source code selection

The first step in constructing the study was to select the *chain expressions* to be used in the study. Hereafter, these are referred to simply as *names* since, taking a functional viewpoint, they *name* a runtime entity. In other words, names are the programing language syntax that identify an object (address) during program execution. (While not studied, an object can have multiple names.) Each name is composed of a collection of parts separated by Java's dot operator. Names include field selectors, expressions denoting a method, and method arguments. One of the parts was selected to be the part of the name that the subject would have to recall.

To balance the time required to take part in the study with the need to collect sufficient data from which to draw statistical conclusions, it was decided that eight questions would be included. Eight questions should encourage initial participation and lead to a high completion rate by the volunteer participants. The eight names used in the eight questions were extracted from production code. Choosing names from production code is important because it improves the study's external validity as the experiment more accurately captures the code used in the engineer's environment; thus, it brings the results closer to those expected when working directly with production source code. However, selecting names from production code represents a compromise between control (e.g., the high-control of artificially constructed source [7]) and applicability of results to the code found in real programs.

The selected names were drawn from a collection of approximately 6.3 million lines of code taken from a cross section of open-source programs (e.g., cvs, jaccounting, cinelerra, sendmail, tomcat, eclipse, jmeter, etc.). In total 508,790 names were extracted. For each, the number of constituent parts (separated by Java's dot operator) was then computed. Fig. 1 presents the distribution of the number of parts per name. For the experiment, those having only one or two parts were dropped because they were too short to create meaningful questions. The authors then scanned the remaining names and selected 200 representative examples. For each of these, the number of syllables was counted.

The final selection of the eight names was based on two factors affecting performance in recalling recently read information. The first is the extent to which the information can be maintained in short term memory. This depends on the short-term memory resources consumed by the encoded information and on other demands on short-term memory resources between the time the information is originally encoded and when it needs to be recalled. The second is influenced by the extent to which the information is already stored in persistent memory. For instance, this information may

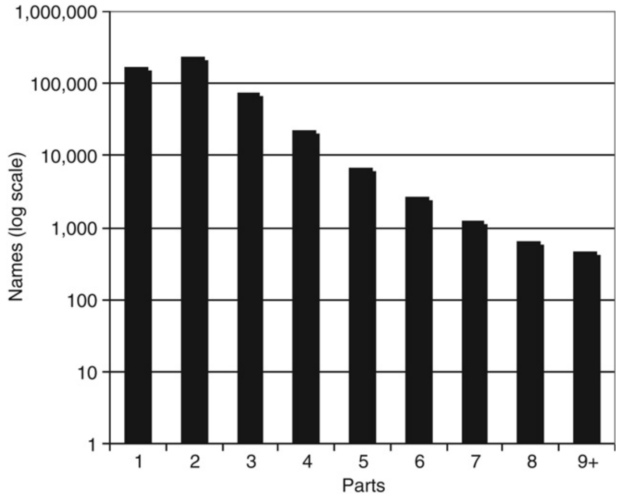


Fig. 1. Extracted names categorized by number of parts (the *y*-axis uses a log scale).

exist because a character sequence has been encountered before or its sound pattern matches (or rhythms with) that of a known word.

These two factors are captured in the study by the variables Length and Ties. Length is used to partition the questions into two groups: *short* (having about 10 syllables and 4 parts) and *long* (having about 20 syllables and 7 parts). The variable Length allows results to be simply stated; for example, "longer names take 20 s more time to study". To provide a finer level of granularity in the models, Length is refined in the study into two counts: the number of parts and the number of syllables.

Ties is used to further partition the questions into those expected to be tied to the subject's persistent memory and those not expected to be so tied. In general, ties come from familiarity with the domain of a program and with programmer experience. Thus, each part of a name has a different strength of tie to persistent memory, and these ties are different for each subject. However, in the experiment, Ties is a binary value that denotes the expectation that the selected part will have ties to all subjects' persistent memory. Ties is true when the selected part comes from the standard Java library. This helps to ensure that the correct value of Ties is assigned to each question. After the assignment was made, it was checked using a search of the World Wide Web. The four parts assumed to have Ties averaged 10 million hits (the query included the search word "java" to avoid false positives). In contrast, the four with no Ties averaged just under 140 thousand (with java getJournalEntry yielding only 7 hits). Having two orders of magnitude more hits makes it reasonable to assume that these parts have Ties to (Java) programmer persistent memory.

From the 200 names, Length and Ties were used to select the eight names and then group them into four categories: short with ties, short with no ties, long with ties, and long with no ties. To create a balanced study, two names from each category were selected. These are shown in Fig. 2, where the underlined part (identifier) is the selected part of the name to be recalled during the experiment. Each entry of the figure also includes the package and file from which the name was extracted.

3.2. Experimental layout

The core of the experiment is laid out as eight questions. Each question is divided into three parts. The first shows a single name to the subject, who is free to study it for as long as desired. Then, as in similar studies, the subject undertakes a distracting activity. In this experiment, subjects were asked to enter the kind of application from which they thought the name might have come. This serves as a distractor, which simulates, for example, the subject considering intervening code. Finally, the subject was shown the name with one of the identifiers elided and asked to enter the missing identifier. The eight questions, shown in Fig. 2, were arranged into two groups (labeled "Group 1" and "Group 2" in the figure). The questions within each group were shown in a random order to avoid systematic learning biases.

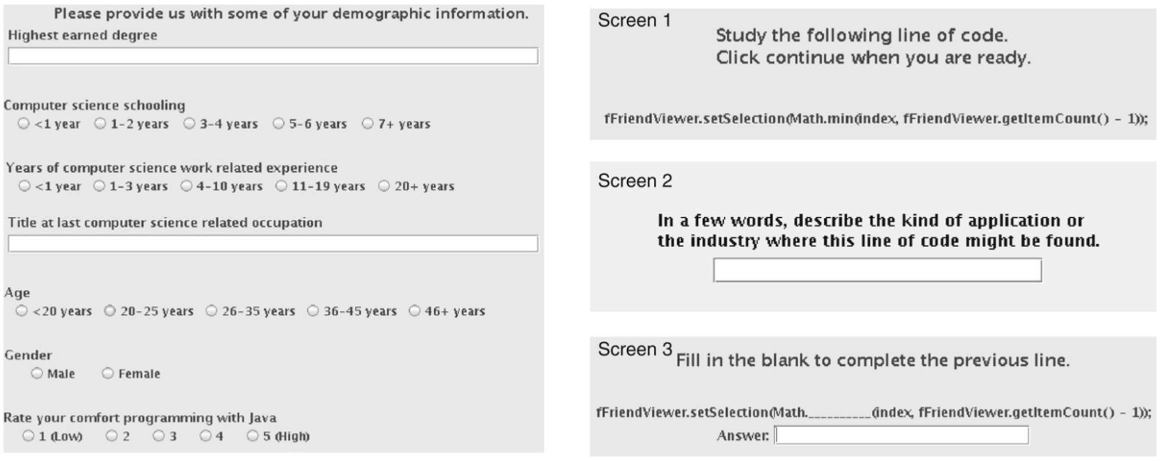
The experiment was conducted over the internet using a Java applet, which has several advantages. First, it allows for wide, quick distribution. Second, the applet prevents the use of the web browser's back button and thus provides flow control. Third, the applet allows the timing of how long subjects spend on each question. Finally, the results gathered are already in a digital format, which supports easy manipulation and statistical analysis and prevents data entry errors.

Before the eight questions, demographic data was collected using the screen shown in the left of Fig. 3. The right of this figure shows one set of the three screens used for each question. A final screen allows participants to provide comments and feedback on the study.

Subjects were recruited via email. A message, which described the study and included the applet's URL, was sent to current students, alumni of several colleges, and various professional groups.

	Syl	Parts
Short		
Ties		
(Group 1) Thread.currentThread().getName().substring(3)	8	5
From jakarta-jmeter/src/functions/org/apache/jmeter/functions/ThreadNumber.java		
(Group 2) request.getQueryString().trim().length()	8	4
From jakarta-tomcat/src/share/org/apache/tomcat/core/RequestDispatcherImpl.java		
No Ties		
(Group 1) digester.log.isDebugEnabled()	10	3
From jakarta-tomcat-connectors/util/java/org/apache/tomcat/util/digester/CallParamRule.java		
(Group 2) payment.getJournalEntry().getTransactions()	11	3
From jaccounting-cvs-090905/.../src/java/com/spaceprogram/accounting/model/RecurrencePage.java		
Long		
Ties		
(Group 1) fFriendViewer.setSelection(Math.min(index, fFriendViewer.getItemCount() - 1))	21	7
From eclipse/pde/internal/ui/editor/plugin/ExportPackageVisibilitySection.java		
(Group 2) context.getDocumentBase().getProtocol().equalsIgnoreCase(Constants.Request.WAR)	21	7
From jakarta-tomcat/src/share/org/apache/tomcat/core/DefaultServlet.java		
No Ties		
(Group 1) manager.getContainer().getLogger().debug(sm.getString(getStoreName())	19	7
From jakarta-tomcat-catalina/.../session/FileStore.java		
(Group 2) paid.compareTo(invoice.getTotal().setScale(2, BigDecimal.ROUND_HALF_UP))	19	7
From jaccounting-cvs-090905/PaymentPage.java		

Fig. 2. The names used in the study's eight questions, categorized by length and the expectation that the selected identifier has ties to persistent memory. The selected identifier is underlined.



Please provide us with some of your demographic information.

Highest earned degree

Computer science schooling
◯ <1 year ◯ 1-2 years ◯ 3-4 years ◯ 5-6 years ◯ 7+ years

Years of computer science work related experience
◯ <1 year ◯ 1-3 years ◯ 4-10 years ◯ 11-19 years ◯ 20+ years

Title at last computer science related occupation

Age
◯ <20 years ◯ 20-25 years ◯ 26-35 years ◯ 36-45 years ◯ 46+ years

Gender
◯ Male ◯ Female

Rate your comfort programming with Java
◯ 1 (Low) ◯ 2 ◯ 3 ◯ 4 ◯ 5 (High)

Screen 1
Study the following line of code.
Click continue when you are ready.

```
fFriendViewer.setSelection(Math.min(index, fFriendViewer.getItemCount() - 1));
```

Screen 2
In a few words, describe the kind of application or the industry where this line of code might be found.

Screen 3 Fill in the blank to complete the previous line.

```
fFriendViewer.setSelection(Math.__________(index, fFriendViewer.getItemCount() - 1));
```

Answer:

Fig. 3. Screen shots of the demographics screen and an example question.

3.3. Variables

The variables collected during the experiment include two response variables and twelve explanatory variables. The first response variable, Time, represents the time spent examining the name shown on the first screen of each question. The second, Correctness, reflects the average judgement of the correctness of the answer on the third screen. Its determination is described below.

The explanatory variables come from two sources: the questions and the demographics. The five variables related to each question include the name's Length (*long* or *short*), Syllables Removed, Syllable Count, Parts, and the existence of Ties to persistent memory. Length and Ties were described previously. The variable Syllables Removed counts the number of syllables in the part selected for the subject to recall. The Syllable Count is the number of syllables in the name, and Parts is counted by separating the name based on Java's dot-operator.

The seven demographic variables are used to consider patterns that arise from who the subjects are. They include the Highest Degree earned, years of CS Schooling, years of Work Experience, Job Title, Age, Sex, and Java Experience (referred to as *comfort* in Fig. 3). This last variable is self-determined by the response to ''Rate your comfort programming with Java'' on a scale of 1–5.

Most variables were collected as categorical variables to eliminate encoding errors. For example, as shown in the left of Fig. 3, numeric values were collected categorically as ranges, limiting free-form data entry problems (e.g., Age entered as "142" or "old").

3.4. Data preparation

Following administration of the experiment, the collected data was prepared for statistical analysis. The processing included categorizing free-form answers, considering the demographic data and comments from the final screen, and removing outliers. Each of these procedures is now discussed in more detail.

Three of the four free-form responses were encoded as categorical variables. The exception was the response to the memory clearing exercise. The first of the three, Highest Degree was categorized using the North American degree titles *high school*, *bachelors*, *masters*, *MBA*, and *Ph.D*. This required mapping some degrees from non-North American participants. Second, the title of the last computer science job was categorized as *student*, *researcher*, *teacher*, *programmer*, or *other* (e.g., *professor* was considered a researcher, *lecturer* a teacher, and *architect* a programmer).

The final free-form response was the value entered for the missing identifier. The result was scored on a scale of 0–4. Exactly correct answers (sans case) were scored a 4 and *almost correct* answers were given a 3. The score 2 was given to answers which had *something right* in them, and 1 to answers that were wrong. Zero was used for answers that were blank (e.g., caused by an errant double click) or where the subject wrote something to the effect that they got distracted. These entries were removed as they represent erroneous conditions.

Scoring was performed separately by two of the authors using a scale from 0 to 4 and then averaging the results. Equivalent answers were grouped together to allow the authors to judge each distinct response once in order to ensure that all responses that were exactly the same received the same score. While not related directly to the study, the number of distinct responses (i.e., the number of groups) per question averaged 28.25 with a minimum of 14 and a maximum 44 distinct responses per question. Statistically, the scores of the two judges were in almost perfect agreement ($\kappa = 0.802$).

Based on information on the final screen or in the demographics, the data for a few subjects was removed. For example, one subject reported writing down each name. A second subject reported being a biology faculty member with little computer science training.

Finally, the time spent viewing Screen 1 was examined. It was decided that responses with times shorter than 1.5 s should be removed because they gave the subject insufficient time to process the code. This affected 18 responses (1.4% of the 1264 responses). In addition, excessively large values were removed. This affected 6 responses (0.5%) each longer than 9 min.

4. Experimental hypotheses

This section presents the five hypotheses studied. Each hypothesis is given an informal name, which is followed by its formal statement and then its motivation. The next section presents the statistical analyses used to investigate each of the five hypotheses.

Hypothesis 1. Length increases study time

$H1_0$: Study Time is the same regardless of Length
$H1_A$: Length increases study Time

The first hypothesis states that longer names require more Time to consider, process, and, thus, understand. All other things being equal, increased Time equates to increased software cost (in particular, software maintenance cost). By itself, this result might not seem a great surprise and the cost might be recovered through improved comprehension, fewer defects, or more efficient change making. Names can be considered too long if any increased costs are not balanced by cost reductions.

Hypothesis 2. Length reduces correctness

$H2_0$: Correctness is the same regardless of Length
$H2_A$: Length reduces Correctness

The second hypothesis states that longer names reduce subjects' ability to recall parts of a name. This is due to the overcrowding of short-term memory; thus, subjects are less able to recall particular parts of the name. An affirmative finding for both Hypotheses 1 and 2 would support the notion that names can be too long.

Hypothesis 3. Memory ties improve correctness

$H3_0$: Ties do not effect Correctness
$H3_A$: Ties improve Correctness

The third hypothesis states that common programming idioms are easier to recall and process because they have Ties to one or more concepts stored in the programmer's persistent memory. Thus, problems that involve more common Java expressions will have greater Correctness [7].

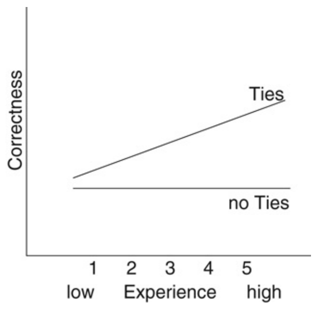


Fig. 4. Hypothesis 5b expected output.

Hypothesis 4. Experience improves correctness

$H4_0$: Java Experience has no effect on Correctness
$H4_A$: Java Experience improves Correctness

The fourth hypothesis essentially reinforces Hypothesis 3 as greater experience is expected to lead to greater persistent memory Ties. Thus, together Hypotheses 3 and 4 support the notion that Java Experience improves Ties to persistent memory that can be exploited in code comprehension and, in this case, lead to higher Correctness.

Hypothesis 5. The impact on correctness of ties to persistent memory, length, and experience are interdependent

$H5_0$: The impact on Correctness of Ties, Length, and Java Experience are independent
$H5_A$: The impact on Correctness of Ties, Length, and Java Experience are interdependent

Hypothesis 5 investigates whether or not Ties to persistent memory, Length, and Java Experience interact with each other. To obtain a clearer picture, this hypothesis is separated into three sub-hypotheses. In each case the hypothesis states the interaction in its 'natural order'; however, each is in fact symmetric. In other words if the impact of *a* increases (or decreases) with *b* then the impact of *b* increases (or decreases) with *a*.

Hypothesis 5a. The impact on correctness of memory ties increases with length

$H5a_0$: The impact on Correctness of Ties to persistent memory is independent of Length
$H5a_A$: The impact on Correctness of Ties to persistent memory increases with Length

Hypothesis 5b. The impact on correctness of experience increases with ties

$H5b_0$: The impact on Correctness of Java Experience is independent of persistent memory Ties
$H5b_A$: The impact on Correctness of Java Experience increases with persistent memory Ties

Hypothesis 5c. The impact on correctness of length decreases with experience

$H5c_0$: The impact on Correctness of Length is independent of Java Experience
$H5c_A$: The impact on Correctness of Length decreases with Java Experience

The first of the three sub-hypotheses, Hypothesis 5a, states that Ties to persistent memory becomes more valuable as short-term memory becomes more crowded. Alternatively, with shorter names, Ties to persistent memory have less value. Hypothesis 5b states that Java Experience becomes more valuable in the presence of Ties to persistent memory. Finally, Hypothesis 5c states that Length becomes less influential with greater Java Experience.

Support for these hypotheses will be evident in the statistical significance of an interaction between the relevant pair of explanatory variables. In a model, this kind of interaction appears as a difference in slopes. For example, as shown in Fig. 4, if the null hypothesis for 5b were rejected, the lines for Ties and no Ties would have different slopes.

5. Experimental results

This section first describes the subject demographics and then, in the first five subsections, it considers the statistical models generated to investigate the five hypotheses. This is followed by a summary of the results, a discussion of threats to validity, and finally, a consideration of several related experiments suggested by the initial analysis. The study included 158 subjects, most of whom (97%) had earned a university degree. In terms of the years of CS Schooling, 7% had 1–2 years, 34%

had 3–4 years, 24% had 5–6 years, and 35% had 7 or more years. The distribution of Age shows that a majority of subjects fell in the 26–35 age range (42%) followed by the 20–25 age range (25%). As would be expected, the years of Work Experience reflect Age with the peak being 34% falling into the 4–10 years category. The other groups were distributed between 13% and 20%. The gender demographics (87% male) shows that the subject population mirrors the field in terms of men and women.

The study did draw more heavy from the research community than would be expected in a purely random sample with 34% researchers, 37% programmers, 15% other, 12% students, and 3% teachers. This bias towards researchers can be expected to be associated with greater schooling. The statistical models show neither of these demographic variables as significant. Hence, neither effect is a significant concern. Finally, the distribution of the self-reported Java Experience was fairly uniform. This rating of subjects' comfort with Java was on a scale from 1 (low) to 5 (high) and breaks down from low to high as 16%, 14%, 18%, 27%, 25%; thus, just over half of the participants rated themselves above the average.

5.1. Hypothesis 1 — Length increases study time

The models considered for each hypothesis start with a collection of explanatory variables. Backwards elimination is then applied to iteratively remove those variables that are not statistically significant. When only a single explanatory variable is of interest, as is the case in the first model, then elimination will either remove the variable indicating no effect or it will retain it, indicating an effect.

After a discussion of the model's statistical significance, each section then considers the practical significance of the variable coefficients in each model. Practical significance concerns the influence of each statistically significant variable in a model. While this judgement is less scientific in nature, it is still useful to consider, in practical terms, the influence of each significant variable.

To investigate the first hypothesis, the initial model includes the explanatory variable Length. This variable is significant ($p < 0.0001$), and indicates that longer names take an average of 20.1 s longer to process. Because of this significance, an attempt to further understand this effect was undertaken. A second mixed-effects model was constructed, this time starting with the explanatory variables Syllable Count and Parts in place of Length. Parts is eliminated; thus, the final model includes only Syllable Count. In this model, it takes an average of 1.80 s longer ($p < 0.0001$) for participants to process each additional syllable. Both models, particularly the more detailed second model, support the alternative hypothesis that as the length of a name increases, the Time required to process the name also increases.

Unlike statistical significance, practical significance requires making a judgement. In this case, does 1.80 s per syllable amount to a meaningful amount of time. Here the answer is clearly yes, especially for activities that consider a large amount of code (e.g., code reviews).

5.2. Hypothesis 2 — Length reduces correctness

Similar to the first hypothesis, the initial model for Hypothesis 2 includes the explanatory variable Length. However, this variable is not significant ($p = 0.778$). To investigate possible reasons for this unexpected result and to parallel the analysis of Hypothesis 1, a second model was constructed beginning with the explanatory variables Syllable Count and Parts. Here both variables are significant: Correctness decreases by 0.0315 with each additional syllable ($p = 0.0019$) and *increases* by 0.0768 for each additional part ($p = 0.0174$).

The interpretation of this second model is complicated by two factors. First, the coefficients of the two variables are opposite in sign (but when tested for, there was no interaction). This, in part, explains Length not being significant. The second complicating factor is that Parts and Syllable Count are (unsurprisingly) highly correlated ($r = 0.859$); thus, names with more Parts also tend to have more syllables. Regarding the first factor, it is relevant to note that the interpretation of coefficients in a statistical model is meant to be done "while holding other variables constant". However, in this case it is not practical to talk about varying Syllable Count while holding Parts constant (or *visa versa*) as the two are so strongly correlated.

Because the model indicates that both factors together are important in predicting Correctness two further models, which consider each of the variables independently, were constructed. Interestingly, both variables' significance decrease ($p = 0.0362$ for Syllable Count alone and $p = 0.5582$ for Parts alone). For Syllable Count alone, Correctness decreases by 0.0109 with each additional syllable. For Parts, the slope is negative, as initially expected, but not significant.

Comparing the coefficient of Syllable Count in the two models where it appears is instructive in gaining an understanding of the impact of the two variables. In the model containing only Syllable Count, the coefficient of Syllable Count is less than in the joint model where Parts helps to dampen the (overzealous) prediction of Syllable Count alone; thus, the combination produces a better prediction. Although more complex than the models for Hypothesis 1, Hypothesis 2 models involving Syllable Count support the alternative hypothesis that Length reduces Correctness (recall). Combined with the first result, longer names are more costly as they take more time to process and lead to lower recall Correctness.

Unlike Hypothesis 1, where the statistical significance is accompanied by practical significance, practical significance is harder to argue for Hypothesis 2. The largest influence is Syllable Count's coefficient of 0.0315; however, here each additional syllable brings only a small decrease: it takes 15 syllables to produce a half a point decreases in Correctness.

Even considering the 95% confidence interval for Syllable Count (0.0118, 0.0513), does not admit the possibility of a large coefficient. One explanation for this is that the names studied are already sufficiently long to overload short-term memory and thus additional syllables over the range studied provide very little additional degradation.

5.3. *Hypothesis* 3 — *Memory ties improve correctness*

The initial model for Hypothesis 3 includes the explanatory variable Ties. This variable is significant in the final model where it increases Correctness by 0.189 ($p = 0.0009$). Because Ties might be affected by the demographic data, a second model was created that includes the demographic variables Highest Degree, CS Schooling, Work Experience, Java Experience, and the interaction Java Experience * Ties. The final model here includes only Ties and Java Experience. Ties has the same coefficient and p values. This indicates the independence of its effect. The impact of Java Experience on Correctness is discussed in the next section. In conclusion, there is statistical support to reject the null hypothesis and conclude that Ties to persistent memory improve recall Correctness. In practical terms, Ties is associated with a fifth of a point increase in Correctness, which indicates that, in addition to statistical significance, Ties has practical significance.

5.4. *Hypothesis* 4 — *Experience improves correctness*

The initial model, for Hypothesis 4 includes the explanatory variable Java Experience. This variable is significant in the final model where it increases Correctness by 0.0849 ($p = 0.0156$) per unit of Java Experience. As with the previous model, the demographic data (e.g., profession or schooling) may influence Java Experience; thus a second model was run that included the demographic variables Highest Degree, CS Schooling, Work Experience. The final model here includes only Java Experience, with the same coefficient and significance. The independence of Java Experience is evident in the second model used when investigating Hypothesis 3, in which only Java Experience and Ties are retained in the final model. As described in Section 5.3, Ties has the same effect as in the simpler models for Hypothesis 3. Attesting to the independence of Java Experience, each unit of Java Experience again increases Correctness by 0.0849 ($p = 0.0156$). Thus, like Ties above, it has a largely independent effect. In conclusion, there is statistical support to reject the null hypothesis and conclude that Java Experience improves recall Correctness. However, in practical terms this effect is not large as Java Experience is measured on a scale from 1 to 5. Thus, over the *entire range* the difference amounts to only one third of a point increase in Correctness.

5.5. *Hypothesis* 5 — *The effect on correctness of ties to persistent memory, length, and experience are interdependent*

5a — *The impact on correctness of memory ties increases with length*
5b — *The impact on correctness of experience increases with ties*
5c — *The impact on correctness of length decreases with experience*.

To test Hypothesis 5, the initial model includes the explanatory variables Length, Java Experience, and Ties, and all three pairwise interaction: Ties * Length, Java Experience * Ties, and Length * Java Experience. After the removal of non-significant terms, the final model includes the three explanatory variables and the interaction Length * Java Experience.

In this model there are two effects on Correctness. First, Ties to persistent memory brings an increase of 0.189 to Correctness ($p = 0.0009$). This means that, independent of other factors, subjects did better with more common Java names. Second, Length and Java Experience take part in an interaction ($p = 0.0336$) (thus their individual impacts on the model cannot be stated separately).

This means that statistically the null hypothesis cannot be rejected for Hypothesis 5a nor 5b; however, for Hypothesis 5c, the null hypothesis can be rejected. The interaction Length * Java Experience is shown in the left of Fig. 5 by the different slopes of the two lines. Here, Java Experience has a greater impact when Length has the value *long* (the line with the greater slope). Symmetrically, to see that the impact of Length decreases with Java Experience observe that the gap between the two lines reduces with increased Java Experience. Interestingly, for those with the most experience, there is no difference in Correctness between short and long names.

Following the example used in previous models, the significance of Length led to the replacement of Length with Syllable Count and Parts in the generation of a new model. The final model includes Java Experience, Ties, Parts, and Parts * Java Experience. Here again there are two effects on Correctness. First, Ties to persistent memory brings an increase of 0.209 to Correctness ($p = 0.0004$). The consistency of the occurrence and coefficient for Ties is indicative of a variable with a strong independent effect. Second, Parts and Java Experience take part in an interaction ($p = 0.0249$).

This again means that, statistically, the null hypothesis can be rejected for only Hypothesis 5c. The interaction Parts * Java Experience is shown in the right of Fig. 5 by the four lines all having different slope (no 6-part names were used in the study). Comparing the two graphs in Fig. 5, the greater detail provided when using Parts is visually evident. Similar to the Length model, Java Experience has a greater impact as the number of Parts increases (appearing as lines with greater slope). Symmetrically, for those with low Java Experience, Parts has a greater impact.

Java Experience is potentially influenced by subject demographics. Thus, a third model was constructed by adding to the prior model the (remaining) demographic variables: Highest Degree, CS Schooling, Work Experience, Job Title, Age, and Sex. This allows the model to determine if more general background or computer science experience plays a role in Correctness.

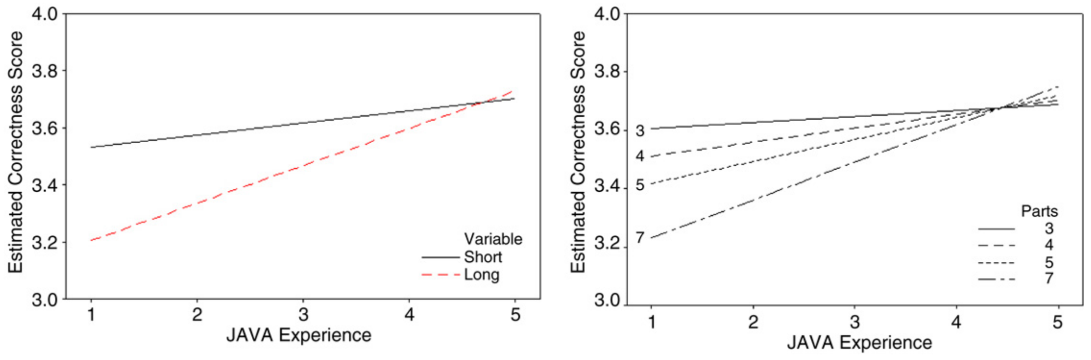


Fig. 5. Interactions between Length and Java Experience and between Parts and Java Experience.

The final model includes Java Experience, Ties, Parts, and Parts * Java Experience, just as before, with the same *p*-values and interpretation. This means that the effects measured in the study are not influenced by the additional demographic factors.

In summary, the interactions Length * Java Experience and Parts * Java Experience provide statistical evidence to reject the null hypothesis of Hypothesis 5c, and conclude that Java Experience's value increases with Length. It is not possible to reject the other two null hypotheses. In short, Ties appears to be of uniform value to all participants. Here it is important to understand that although one might expect a positive correlation would exist between Java Experience and Ties, this is not supported by the data. In this case the result is likely an artifact of the study design. As can be seen in Fig. 2, the questions having Ties involve common methods from the Java library. Thus for these questions all participants had sufficient Java Experience to gain benefit from Ties. To fully test the validity of this observation would require the design of a new study that included parts (identifiers) expected to be known only by programmers who have attained a certain level of Java Experience. It is possible, given the data collected, to check if the interaction between Length and Java Experience is *masking* an interaction between Ties and Java Experience. This is done by simply omitting Length * Java Experience from the initial model and re-running the analysis. Doing so, the final model includes only Java Experience and Ties. Being devoid of interactions supports the observation above that Ties brings the same benefit to all subjects and is not having an interaction masked by Length * Java Experience.

5.6. Implication of main study results

This section summarizes the results of the study and relates the results to program comprehension during maintenance and evolution and during initial code construction. In particular, program comprehension activities include inspection, maintenance, and extension of existing software systems. The next section considers several additional questions that were not addressed in the main hypotheses.

In summary, the number of syllables in an identifier greatly influences the time required for a programmer to comprehend source code and, to a lesser extent, the correctness with which a programmer can recall a particular part (identifier). More commonly used identifiers, by having increased ties to persistent memory, and greater experience programming in Java improve recall correctness. Furthermore, as the length of a name increases, Java experience has a greater influence on the performance of a programmer. Finally, no demographic information apart from experience with Java was found to be significant.

Combined, these results reinforce past proposals advocating the use of limited, consistent, and systematic vocabulary in identifier names [1,3,6,9,15]. From Hypotheses 1 and 2, 'excess' length negatively impacts time and recall correctness. Furthermore, care in choosing a consistent vocabulary, and refraining from needlessly adding to the vocabulary, is emphasized by the support for Hypotheses 3 and 4. Together, good naming principles limit length and reduce the need for specialized vocabulary. Finally, the support for Hypotheses 1, 2 and 5c indicates that to a limited extent programmers can be trained to handle length related issues; however, this inevitably consumes mental resources that might be better spent on other activities.

The initial activity undertaken by an engineer when making a maintenance fix or enhancement to a program, is to understand (at least the part of) the program being manipulated. This inevitably involves reading parts of the source code. It is this aspect of program comprehension that is the study's primary interest. In particular, following the adage *time is money*, longer reading time for code translates directly into increased cost.

During initial code construction the primary source code reading activities include reading one's own code and the examination of third party libraries. The author of a code can expect to take advantage of ties to persistent memory, (unless they choose particularly bad names). Thus, the author stands the best chance of benefiting from well-chosen names that are not too long and include sufficient ties to persistent memory.

Finally, when using (unfamiliar) third-party library code, programmers can make use of an IDE's auto-complete function to learn the library code. However, auto-complete is not a substitute for good upfront library name design. For example, accepting an auto-complete proposal can lead to using the wrong method. One implication of the study's results is that this kind of mistake is more likely in the presence of longer names. Thus, the results serve to reinforce the push for concise and consistent names [6]. Another downside to auto-complete is that it makes de facto abbreviations out of identifier prefixes. Lacking planning, these are unlikely to be good abbreviations.

5.7. Further analysis

Although the main hypotheses were addressed in the discussion above, their consideration raises several additional questions in addition to the five hypotheses. This section examines four such questions.

(1) What is the impact of Syllables Count **when** Parts **is held constant?**
Although the study did not hold Parts constant, the answer can be obtained using statistical techniques. In this case, backward elimination produces models that estimate the effect of each variable while controlling for the other variables in the model. For instance, in Section 5.1 the second model includes both Parts and Syllable Count. In this new model where Parts is held constant, it takes an average of 1.80 s longer for participants to process each additional syllable.

(2) What impact does Prefixing Parts **(defined below) have on** Correctness**?**
For many comprehension activities undertaken by programmers, the method being called is more important than its arguments. To investigate the influence of the method name, an additional variable, Prefixing Parts, was computed. For example, the name account.invoice.setScale(Decimal.ROUNDUP) has 5 parts, but only 3 Prefixing Parts. The analysis for all Hypotheses involving Length were then re-run with Length and Prefixing Parts used in place of just Length. In all models Prefixing Parts and interaction terms involving it dropped out. An explanation for this is that participants were told to study the entire name and not just the prefix; programmers trying to get an initial understanding of the code often focus on the method names and only later the actual parameters. Therefore, it is possible that in a more general compression task, Prefixing Parts would prove significant.

(3) Is greater Time **associated with increased** Correctness?
The third question is motivated by the observation that verbal story tellers often do not exactly reproduce a story. This is because it is sufficient to recall a general concept and then fill in words that were not recalled exactly. In light of this observation, participants in this study who spent longer examining a name might develop a better understanding of the name and thus be able to *reconstruct* it even if the exact missing part was not available in their short-term memory.

The data collected allows this effect to be tested. In particular, Correctness is not a simple boolean variable, but rather was rated on a scale from 0 to 4. Informally, looking at the incorrect but close answers, it is possible that some were reconstructions. To test for this possibility formally, a model predicting Correctness using Time as the sole explanatory variable was constructed. In this model, Time is *not* statistically significant. To reinforce this result, the model was reconstructed ignoring perfectly correct answers. In this case, Time might play a role for those participants who have to perform reconstruction, but not those who could precisely recall the removed part. However, Time is not significant in this model either. Finally, a complete backward elimination was performed including the other explanatory variables. Again Time drops out of the model and is thus not statistically significant.

This result does not mean that a better understanding of when a programmer is reconstructing and when she is recalling is uninteresting. Indeed it is quite plausible that more experienced programmers rely heavily on reconstruction. To correctly test for this effect would require a study specifically designed to isolate the effects of recall and reconstruction. Such a study is left to future work.

(4) Does the length of the removed portion of the name play a significant role in explaining Correctness**?**
In order to examine this question, additional models were constructed adding Syllables Removed wherever Syllable Count appeared (including interactions) for hypotheses concerning Correctness. A model for Hypothesis 1, which concerns Time spent on Screen 1, was not constructed because Syllables Removed, which is unknown during the reading of Screen 1 cannot logically effect Time. Although Syllables Removed was not controlled in the design of the study, these lengths for the shorter and longer names are very similar. A total of 14 syllables were removed from the short questions and a total of 12 from the long questions. As seen earlier in this section, larger values of Syllable Count were more likely to overload participant short-term memory. The impact of Syllables Removed would be expected to come from longer removed words being hard to reconstruct. Thus, both variables potentially play a role in the new models.

The implications of these changes on each Correctness hypothesis are now separately considered. For Hypothesis 2, starting with Syllable Count, Parts, and Syllables Removed, all three are present in the final model. Including Syllables Removed improves the model for Hypothesis 2 dramatically (AIC decreases from 1890.1 to 1848.6). In the final model Correctness decreases by 0.1016 with each additional syllable ($p < 0.0001$) and by 0.1562 with each syllable in the removed part ($p < 0.0001$), but again *increases*, now by 0.3251, for each additional part ($p < 0.0001$). The interpretation of the signs of the coefficients is the same with Parts dampening an over-approximation obtained using Syllable Count and, in this case, Syllables Removed.

Thus, the new model for Hypothesis 2 is better. All variables have higher significance, and AIC is smaller. This improvement indicates that the original model was missing important factors. In this case, after accounting for Syllables Removed, both other variables become more statistically significant as well as have larger (absolute) values.

The models for Hypotheses 3 and 4 do not involve Length and are thus unchanged. For Hypothesis 5, the new model starts with the variables Parts, Ties, Java Experience, Syllables Removed, Syllable Count, and all pair-wise interactions. Unfortunately, the design of the original study does not provide sufficient data to estimate all of the interactions simultaneously. Thus interactions must be prioritized. In this case those that related to the Hypothesis (e.g., Ties * Syllables Removed) are preferred over others (e.g., Parts * Syllables Removed, which involve two length measures). The largest initial model that was feasible to run started with the interactions Syllable Removed * Java Experience, Syllable Removed * Ties, Syllable Removed * Parts, Syllable Removed * Syllable Count, Parts * Java Experience, Syllable Count * Java Experience, and Ties * Java Experience. Separately testing the significance of the remaining three interactions Parts * Ties and Parts * Syllable Count are far from being significant, while Syllable Count * Ties, with a p value of 0.0466, is just barely significant. All three are therefore assumed to not be significant. The final model includes the three interactions Syllable Removed * Ties, Syllable Removed * Syllable Count, and Syllable Count * Java Experience. As with the models from Section 5.5 there is support for Hypothesis 5c. This time from the interaction Syllable Count and Java Experience rather than the interaction between Parts and Java Experience. Nonetheless, both models include support for the interaction between some measure of length and Java Experience. In addition, the new model shows support for Hypothesis 5a in the interaction Syllables Removed * Ties. As with the other new model, this model is a far better model (AIC drops from 1880.2 to 1835.1, a considerable drop). This is perhaps unsurprising because it has more statistically significant terms. It is this improvement in descriptive capability of the model obtained by incorporating Syllables Removed that allows the interactions with Ties to participate in the final revised model.

5.8. *Threats to validity*

There are four threats relevant to this research: external validity, internal validity, construct validity, and statistical conclusion validity. External validity, sometimes referred to as selection validity, is the degree to which the findings can be generalized to other (external) settings. In this experiment, selection bias is possible in the selected names; thus, results from the experiment may not be applicable to other source code. Careful selection of code from a large code base mitigates the impact of any such bias. It is also possible that taking the code out of context changes the recall process and thus the results when reading code in context would differ. Inclusion of the memory clearing step was added to simulate the subject looking at related code. Finally, selection bias is also possible in the selection of participants, which is clearly a convenience sample. However, the demographic variables were not statistically significant, which indicates the absence of a problem related to the subject sampling.

Second, there exist several threats to internal validity, the degree to which conclusions can be drawn about the causal effect of the explanatory variable on the response variables. First, statistical associations do not imply causation; though, given the experimental setup, one should be able to infer that differences in performance are due to the explanatory variables considered. Second, learning effects occur when subjects believe that they have detected a pattern in the problems and attempt to exploit it to improve their performance on later problems. Questions were shown using a random order to avoid any systematic learning bias. Furthermore, the independence of the questions reduces the ability of subject to learn from past problems. Finally, some threats to internal validity are not easily controlled during an internet study. For example, there is no good way to prevent subjects from taking notes nor is it easy to determine their level of motivation. Subjects being volunteers and not receiving any reward (in particular any performance-based reward) should minimize these threats. Other potential threats to internal validity, for example, history effects, attrition, and subject maturation [2] are non-issues given the short duration of the experiment.

Construct validity assesses the degree to which the variables used in the study accurately measure the concepts they purport to measure. Most of the study variables (e.g., Length and Age) can be measured precisely. This threat is only a concern for the variables Ties and Parts. For Ties the authors assessment of which parts are tied to persistent memory might differ form those with actual Ties. The web hits reported in Section 3.1 provide a measure of unbiased confirmation that the parts assumed to have ties to persistent memory were more common and thus more likely to have such ties. Parts may represent an over estimate because of *chunking* (e.g., Math.min may be processed as a single *chunk* rather than two parts. Chunking is a common trick used by the brain to extend limited short term memory resources. Finally, a threat to statistical conclusion validity arises when inappropriate statistical tests are used or when violations of statistical assumptions occur. The models applied are appropriate for analyzing unbalanced, repeated-measures data, so that the conclusions drawn from the statistics should be valid.

6. Related work

Related work comes on two 'sides' of this study: at a 'lower' level there is research on human memory and cognition and at a 'higher' level there is research on the impact of naming on programmer comprehension. To begin with, New et al. study the impact of word length on lexical decision latencies [14]. The study considered words that range from 3 letters to 13 letters, which, unlike its predecessors, represents the entire range of normally encountered English word lengths. They found that the number of syllables, letters, and, orthographic neighbors, each made significant independent contribution toward lexical decision latencies. For example, each syllable added 20 ms to the time needed to recognize a word. In comparison, subjects in this study were not simply reading, but studying code; thus, it is unsurprising that each syllable added 1.65 s (1650 ms) to the study time.

In terms of letters, the analysis revealed an unexpected U-shaped curve — decision latencies were greater for short and long words: word length was facilitatory from 3–5 letters, null from 5–8 letters, and inhibitory from 8–13 letters. This pattern existed even within the 12,987 bisyllabic words; thus, the U-shaped curve is not a confound of the number of syllables. In fact, the inhibitory effect of number of letters is robust in that it is not a confound of any other lexical factors. The U-shaped result potentially explains the mixed results of past studies (twelve summarized studies find length to be *either* inhibitory *or* null in effect) probably because these studies used restricted length ranges.

One explanation offered for this phenomenon is that the reading of words with a length of 6–9 letters has the highest chance of being done in a single fixation. Shorter words are often skipped and longer words must be re-fixated as letters are more difficult to perceive the farther they are from the fixation point. In addition, parallel word processing is observed only in a small region of the visual field, with highly skilled readers, and with a familiar font.

In terms of lexical decision latencies, most Java names are quite long because camelCasing provides insufficient separation (e.g., the identifier equalsIgnoreCase is read as a single sixteen letter word). Thus, they are well into the inhibitory range in terms of decision latency. This result underlies the study presented herein. It provides a general understanding of human lexical processing.

New et al. conclude that "the relationship between word length and lexical decision times is less straightforward than has been assumed". The models presented in Section 5 indicate that reading of source code by programmers is also less straightforward than might be assumed.

The second related experiment considers the consequences of limited short-term memory capacity on subjects' performance comprehending short artificial code sequences [7]. This study provides results regarding the impact that different character sequences have on the cognitive resources required during program comprehension. Two identifier attributes were studied: the amount of short-term memory required to hold an identifier's spoken form (i.e., the number of syllables) and pre-existence (i.e., ties to persistent memory). Thus, identifiers belonged to one of four possible sets: a single character whose spoken form contained a single syllable (short, no ties), an English word whose spoken form contained one syllable (short, ties), three characters not forming a word whose spoken form contained three syllables (long, no ties), and a word whose spoken form contained three syllables (long, ties).

This study found that processing character sequences is effected by the kind of sequence. In terms of correct answers, one syllable words received the most correct answers. This corresponds to short names with ties to persistent memory. The worst results were for three unrelated letters, which corresponds to long names with no ties to persistent memory and provides evidence of the crowding effect. Thus frequently used character sequences (i.e., words) are recognized faster and are more readily recalled than rare ones. In addition, many performance characteristics are slower and more error prone for non-words compared to words.

In comparison, the study laid out in Section 3, is less *artificial* in that the code came from production programs rather than being constructed to satisfy initial experimental conditions. In exchange, the experimental conditions are not as tightly controlled. For example, the number of syllables for long names varied from 19 to 21. This makes the results harder to analyze and interpret, but places the result closer to the experience of programmers.

The third study considers the work of Liblit et al., which aims to relate standard programming practice to modern theories of human language and cognition [11]. The authors find that "programmers use names in regular, systematic ways that reflect deep cognitive and linguistic influences. This allows names to carry [natural language] semantic cues that aid in program understanding and support software development. However, overuse of abbreviations can lead to a preponderance of unique symbols programmers must decipher [8], which may lead to inhibited understanding [11].

Liblit et al. observe, but do not study, that "if names are informative, then longer names, with more embedded sub-words, should be more informative. Yet there are practical limits to the lengths of names, just as there are practical limits to the lengths of words in natural language. Longer names may be more informative, but they are also more cumbersome to read". As an example, they present the two snippets shown in Fig. 6 that each calculate Euclidean distance. In one, identifiers have names such as distance_between_abscissae and first_ordinate, and in the other, identifiers have the more traditional names such as dx and y1. Of the two, Snippet 1 has the longer more informative identifiers. In contrast, their lengths require multiple fixations and more quickly overcrowd short-term memory; thus, requiring re-fixating and greater concentration. For a reader familiar with the abbreviations used in Snippet 2, its shorter names are easier to process as it requires less effort. However, imagine reading Snippet 2 without previously knowing the Euclidean distance formula. Most programmers can recall this kind of situation in which a fragment of code was meaningless until the semantics of the names became known.

Liblit et al. conclude that terse and abbreviated names enhance readability at the expense of *expressiveness*. The results from Section 5 dampen the expressiveness advantage by studying the memory demand of longer names. Similar to lexical decision latency studied by New et al., this suggests a U-shaped comprehension result in which names that are too short inhibit comprehension by failing to be sufficiently expressive, while those that are too long inhibit comprehension by overtaxing programmer short-term memory; thus, the results from Section 5 combined with those of Liblit et al. extend the results of New et al. to software.

Other researchers have studied naming's impact on comprehension at a *higher level*. For example, Takang et al. investigate the hypothesis that programs with full-word identifiers are more understandable than those with abbreviations [16]. While

Snippet 1
```
distance_between_abscissae = first_abscissa - second_abscissa;
distance_between_ordinates = first_ordinate - second_ordinate;
cartesian_distance = square_root(
   distance_between_abscissae * distance_between_abscissae
   + distance_between_ordinates * distance_between_ordinates
);
```

Snippet 2
```
dx = x1 - x2;
dy = y1 - y2;
dist = sqrt(dx * dx + dy * dy);
```

Fig. 6. Comparison of informative versus easy-to-read names.

a surprise to the authors, this hypothesis is not supported by the quantitative data collected. The study from Section 5 may shed some light on this unexpected result as the longer names may have overcrowded subject's short-term memory.

The work of Deissenböck and Pizka is also of interest as it presents a formal model based on bijective mappings between concepts and names for adequate identifier naming [6]. The model is based on the observation that "research on the cognitive processes of language and text understanding shows that it is the semantics inherent to words that determine the comprehension process". It includes rules for the *correct* and *concise* naming of identifiers. The core idea is that within a given program a concept should always be referred to by the same name. Their goal is that names contain enough information for an engineer to comprehend the precise concept involved without too large a strain on short-term memory.

The challenge that the results from Section 5 raise for keeping a one-to-one relationship between names and concepts comes from the need to differentiate concepts normally leading to longer names. For example, a program with a single position concept (e.g., absolute positions) might use the variable position. If later evolution of the code requires the concept of relative positions, then the names need to differentiate these two. In this case relativePossitions and absolutePossitions are sufficient. However, this is not a sustainable pattern. For example, the program eMule includes the method name GetFileTypeDisplayStrFromED2KFileType. The length of this name highlights an attempt to capture a concise concept. However, this was clearly done at the cost of readability making it an awkward choice.

Next, Laitinen studied the impact of *Natural Naming* by introducing a method for estimating the understandability of source code [8]. The core idea of this work is that every program has its own unique language made up of the technical and textual symbols found in the program. Programs contain symbols that cannot be considered to belong to the programming language. For example, consider a C function named "ispdrome". The symbol "ispdrome" does not belong to the C programming language (it is not mentioned in the C standard). Neither can we say that the symbol "‘ispdrome" belongs to some natural language (it is not in the English dictionary). Source that avoids unnecessary symbols is considered better.

Laitinen treats languages as sets of symbols and defines the understandability of a program as proportional to how understandable the language it contains is. This is inversely proportional to how complex the language is. Two rules are defined to estimate the understandability of a language [8].

- Rule I: Smaller languages are usually easier to understand than larger languages: the number of symbols in a language affects its understandability.
- Rule II: It is easier to understand closely related languages than more distantly related languages. Thus, a person who has mastered, for example, Italian would find French easier to understand than Mandarin.

The key result of a study of four programs showed that programs using abbreviations have larger languages that grow faster as the program matures. Laitinen concludes that not allowing unnecessary symbols to enter the language (e.g., abbreviations) can improve the programming process by controlling linguistic complexity during software development. He thus advocates avoiding abbreviations or other unnecessary symbols and thus controlling the linguistic process during software development.

7. Future work

This paper reports on an experiment using single lines of production code. It thus builds on similar past experiments, most notably that of Jones [7] who use artificial code and Lawrie et al [9,10] who artificially manipulated the identifiers in production code. The next step in this direct line will use the experience obtained from conducting this experiment to consider increasingly realistic environments. While considerably more complex to set up and administer, the next experiment will consider larger syntactic units (e.g., individual methods) extracted from production code.

To help plan such a study, a prototype was recently conducted using an applet over the internet. It involved 89 participants. While a smaller group, the demographic breakdown of the 89 is essentially the same as the study reported on in this paper.

The task involved debugging one of three versions of a function. The versions differ only in the length of their identifiers: short (abbreviated), full word, and phrase (artificially long identifiers). The full-word variant is shown in Fig. 7 from which the level of complexity in the task can be gauged. This preliminary version required the creation of two alternate versions

```
// This method is supposed to calculate the interest that could be earned on an investment when
// interest is compounded daily, monthly or annually. However, this method actually calculates
// simple interest based upon the principle each time. It should be compounded and added at the
// time frame specified.

public double calculateCompoundInterest(double principle, double interestRate, int compoundOption)
{
   final int DAILY = 0;
   final int MONTHLY = 1;
   int totalPayments;

   double newBalance = principle;

   switch(compoundOption)
   {
      case DAILY: totalPayments = 365; break;
      case MONTHLY: totalPayments = 12; break;
      default: totalPayments = 1;
   }

   double appliedInterestRate = (interestRate / (100 * totalPayments));

   for(int currentPayment = 0; currentPayment < totalPayments; currentPayment++)
   {
      newBalance += (principle*appliedInterestRate);
   }

   return newBalance - principle;
}
```

Fig. 7. An example problem from the pilot function debugging study. Participants were shown one of three version of the code: one using abbreviations, one using full words (shown here), and one using elongated phrases. In this example the assignment in the for loop should read newBalance += (newBalance*appliedInterestRate).

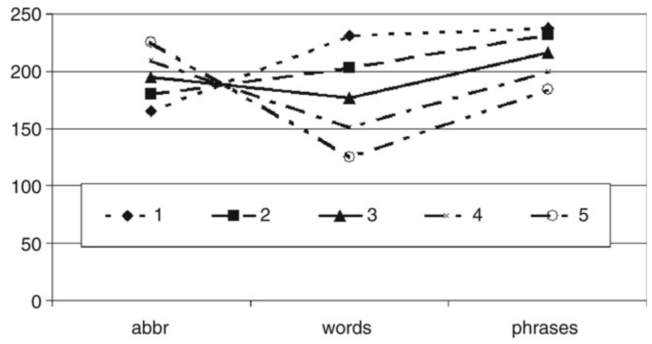


Fig. 8. Data from debugging experiment.

of each function. This allows control on the complexity and the differences between the versions and focuses the difference on the identifiers.

Fig. 8 summarizes data collected from the preliminary study. Here the *y*-axis shows time in seconds on the debugging exercise and the *x*-axis shows the three variants. Each line represents a level of subject experience from 1 (low) to 5 (high). Two trends are evident from the graph.

First, from words to artificially long phrases all five different experience levels show an increase in time taken; thus, reinforcing the notion that, at least when debugging, names can be too long. It is interesting to note that the impact is less pronounced for the least experienced (the slope of the top line is lower). Further study is needed to understand this affect. The second, and far more interesting, observation comes from the trend between the abbreviated names and those constructed from full words. Here there appears to be an interaction between length and experience with length aiding experienced programmers (experience levels 3, 4, and 5 shows a reduction in time taken to perform the debugging task). However, increased length hurt inexperienced programmers. Further study of this affect is warranted.

Presentation and discussion of the study from Section 5 has often led to a discussion concerning the impact of camelCasing on program comprehension. Thus a follow up study comparing the comprehension correctness and speed using camelCasing (asInJava) and underscores (as_in_C) is planned. Researchers who study the reading of natural language, observe that identifiers constructed using camelCasing would be read as one long word, but the use of underscores would be read more quickly as two words (e.g., compare polarPoint with polar_point). This raises the question: will this hold true for programmers or can programmers be trained to read camelCasing as quickly as multiple words?

Finally, persistent memory has two dominant modes: spatial and sequential. Spatial recall is limited to approximately 7 entities; thus, for example, limiting the size and complexity of useful UML diagrams. In contrast, much larger (longer) material can be stored in sequential memory (e.g., the words or notes of a song or blocks of source code). Finding an appropriate mix of these two and exploiting it in comprehension activities represent more distant future research.

8. Summary

The study described in this paper shows that names used in existing production code are long enough to crowd programmer's short-term memory. This provides evidence that software engineers need to consider shorter, more concise names [6]. As the study considers individual names extracted from production code, it tends to underestimate the demand on memory because there is no need to remember context as well. At a minimum this study shows that results from general reading comprehension apply to programmers reading code. More generally, the study extends prior results [7] from a highly controlled setting to a less controlled setting that, by using production code, more accurately reflects programmer behavior.

Acknowledgments

Special thanks go to all the participants as this work would not have been possible without their help. Thanks are also due to John Blair and Matt Hearn for their help in setting up the survey and to Mary Jo Coiro for help with aspects related to memory and comprehension. Finally, we thank the anonymous reviewers for their careful reading of the experiment and suggestions for improvements to this paper and for related experiments to be considered. Also we thank the ICPC participants for their conversation, suggestions, and suggested clarifications. This work is supported by NSF grant CCR-0305330.

References

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, E. Merlo, Recovering traceability links between code and documentation, IEEE Transactions on Software Engineering 28 (10) (2002).
[2] D. Sjøberg, J. Hannay, O. Hansen, V. Kampenes, A. Karahasanovic, N. Liborg, A. Rekdal, A survey of controlled experiments in software engineering, IEEE Transactions on Software Engineering 19 (4) (1993).
[3] B. Caprile, P. Tonella, Restructuring program identifier names, in: ICSM, 2000.
[4] The Psychological Corporation, WAIS-III WMS-III Technical Manual, Harcourt Bract and company, 1997.
[5] N. Cowan, The magical number 4 in short-term memory: A reconsideration of mental storage capacity, Behavioral and Brain Sciences 24 (1) (2001).
[6] F. Deißenböck, M. Pizka, Concise and consistent naming, in: Proceedings of the 13th International Workshop on Program Comprehension, IWPC 2005, St. Louis, MO, USA, May 2005.
[7] D. Jones, Memory for a short sequence of assignment statements. C Vu, 16 (6) (2004).
[8] Kari Laitinen, Estimating understandability of software documents, SIGSOFT Software Engineering Notes 21 (4) (1996).
[9] D. Lawrie, C. Morrell, H. Feild, D. Binkley, What's in a name? A study of identifiers, in: 14th International Conference on Program Comprehension, 2006.
[10] D. Lawrie, C. Morrell, H. Feild, D. Binkley, Effective identifier names for comprehension and memory, NASA Journal of Innovations in Systems and Software Engineering 3 (4) (2007).
[11] B. Liblit, A. Begel, E. Sweetser, Cognitive perspectives on the role of naming in computer programs, in: 8th Annual Psychology of Programming Workshop, Brighton, UK, September 2006.
[12] C. Morrell, J. Pearson, L. Brant, Linear transformation of linear mixed effects models, The American Statistician 51 (1997).
[13] J.S. Nevid, Psychology Concepts and Applications, Houghton Mifflin Company, Boston, MA, 2003.
[14] B. New, L. Ferrand, C. Pallier, M. Brysbaert, Reexamining the word length effect in visual word recognition: New evidence from the English Lexicon Project, Psychonomic Bulletin & Review 13 (1) (2006).
[15] J. Rilling, T. Klemola, Identifying comprehension bottlenecks using program slicing and cognitive complexity metrics, in: Proceedings of the 11th IEEE International Workshop on Program Comprehension, Portland, Oregon, USA, May 2003.
[16] A. Takang, P. Grubb, R. Macredie, The effects of comments and identifier names on program comprehensibility: An experiential study, Journal of Program Languages 4 (3) (1996).
[17] G. Verbeke, G. Molenberghs, Linear Mixed Models for Longitudinal Data, second ed., Springer-Verlag, New York, 2001.