An Empirical Comparison of the Accuracy Rates of Novices using the Quorum, Perl, and Randomo Programming Languages

Andreas Stefik Susanna Siebert Melissa Stefik Kim Slattery
Department of Computer Science
Southern Illinois University Edwardsville
Edwardsville, IL 62026-1656
{stefika, susanna.siebert, stefikm, slattery.kim}@gmail.com

Abstract

We present here an empirical study comparing the accuracy rates of novices writing software in three programming languages: Quorum, Perl, and Randomo. The first language, Quorum, we call an evidence-based programming language, where the syntax, semantics, and API designs change in correspondence to the latest academic research and literature on programming language usability. Second, while Perl is well known, we call Randomo a Placebo-language, where some of the syntax was chosen with a random number generator and the ASCII table. We compared novices that were programming for the first time using each of these languages, testing how accurately they could write simple programs using common program constructs (e.g., loops, conditionals, functions, variables, parameters). Results showed that while Quorum users were afforded significantly greater accuracy compared to those using Perl and Randomo, Perl users were unable to write programs more accurately than those using a language designed by chance.

Categories and Subject Descriptors D.3 [*Programming Languages*]; H.1.2 [*Information Systems*]: User/Machine Systems — Software Psychology

General Terms Design, Experimentation, Human Factors, Standardization

Keywords Programming Languages, Intuitiveness, Comprehension, Language Reductionism

1. Introduction

How to design a programming language that is sufficiently easy to use for all people, from beginners starting their journey in computer science courses to professionals creating globally distributed products, is neither obvious nor easy. For decades, computer scientists have argued, largely anecdotally, about what makes a language robust, elegant, or easy to use. While debate is common, the types of sophisticated experimental designs and thorough empirical data common in social disciplines like psychology have largely not been adopted by the programming language design community. As such, our community has, as Hanenberg poetically states, put far too much faith, hope, and love [3] in language design.

In this paper, we present an objective test of the accuracy by which novices can program a computer in three computer programming languages, 1) Quorum (previously called Hop [14]), 2) Perl, and 3) Randomo. In doing so, we present a novel adaptation of an accuracy measurement paradigm called Artifact Encoding [15] and define two novel computer programming languages. In the first language, Quorum, we have used survey methods [14], usability studies, and field tests [16] to help us design the programming language and make it easier to use. We call such a language an *Evidence-Based Programming Language* in the sense that we have continuously altered the syntax, semantics, and API design according to both our observational work with novices and results from the literature [6, 11, 18].

During the course of these studies, we have observed that novices learning to program at the university or younger levels (and also blind children [16]), can have significant difficulty learning the syntax of general purpose programming languages, which may initially seem arbitrary (e.g., ||, ^, @_[0]). Given this observation, novices may *think* programming language syntax is rather arbitrary, but how accurately could novices use a programming language designed by a random number generator? We designed such a language: Randomo. If the reader will excuse a metaphor, we call languages designed in such a way *Placebo languages*. We propose that Placebo languages may be useful as an objective form of control group in language design studies.

These observations and language designs led us to ask the following **research question**: *Can novices using pro-*

gramming languages for the first time write simple computer programs more accurately using alternative programming languages? To analyze the issue, we conducted a study with a repeated-measures between-subjects design comparing accuracy rates of novices using common syntax (e.g., loops, conditionals, functions, variables, parameters) in the languages Quorum, Perl, and Randomo. Results show that Evidence-Based Programming Languages afford accuracy rates that are statistically significantly higher amongst novices compared to those using Perl or Randomo, as measured by Artifact Encoding [15]. Novices using Perl were unable to write programs significantly more accurately than Randomo, our Placebo language.

In the course of this paper, we will first discuss related work. Then, we will move to the details of our experimental procedure, including a brief explanation of Artifact Encoding. Then we will discuss our results, their potential implications, threats to validity, and conclude.

2. Related Work

As programming itself is such a significant component of what computer scientists do, its study in the literature is only natural and has a rich history. While a complete review is outside the scope of this work, in this section, we review some of the major themes related to the usability of programming languages.

The empirical studies and programming tools presented here have been heavily influenced by several lines of research on programming language usability. For example, McIver focused her research on the programming language GRAIL in usability studies, making specific predictions about what kind of syntax and semantics would be sensible [10]. One specific suggestion was that the unicode arrow might be more intuitive to novices for assignment (e.g., compared to =, :=, or other common operators). While McIver's hunch is contradicted by one of our previous empirical studies [14], the low level thinking on specific constructs used in GRAIL heavily influenced the bottom-up approach used in the Quorum programming language.

Other language designers have also attempted to make languages intuitive or easy to use. For example, Holt et al. created the teaching programming languages SP/k and Turing [4]. Similarly, in the late 80's and early 90's, many researchers focused on analyzing either the comprehension of languages [12] or qualitative analysis of a specific language (see Taylor for one of many examples [19]). While work like Taylor's is qualitatively interesting, it does not make specific predictions about how to *change* programming languages to improve them, nor does it provide empirical data to support its claims. For a systematic review of novice programming systems, see Kelleher and Pausch [7].

Many authors in recent years interested in how novices or end-users might program have been influenced by the visualization or multimedia literature [1, 6, 8]. While such systems are interesting, quantitative experiments indicate that they are not a silver bullet. For example, Garlick and Cankaya [2] showed that students beginning with Alice had statistically significantly lower grades than those starting with pseudo-code in introductory computer science courses. Further, novice programming systems are typically only used at the beginning of a computer science curriculum, if at all, while general purpose programming languages are typically used beyond that point and in industry.

While many lessons can be learned from these systems, the literature suffers from at least two significant drawbacks: 1) work conducted on programming languages is sometimes less empirically rigorous than it should have been, and 2) a significant amount of analysis was conducted on the algorithmic performance of programming languages, but dramatically less on the humans that use them. Markstrum's argument is salient [9]. He observed that many programming language papers tend to make claims about their language without backing them up with empirical data. Consider, for example, the language Turing. While Holt and Cordy spent considerable effort attempting to make their language easier to use for novices, the only published data collection on the language appears to have been a survey comparing the language to Fortran. Further, in our personal correspondence with Holt and Cordy, data was collected internally by their research team on novice use of Turing, but this work was never published and has been lost to history. In summary, there is little empirically rigorous work in the literature on general purpose programming language design.

On the second point, Hanenberg [3] argues that human factors and empiricism should be included in the study of programming languages. Fortunately, the modern literature has picked up steam in this area [11], influencing the current research. Specifically, APIs in the Quorum programming language are being designed according to the empirically refined directions in Stylos [18], Quorum does not allow constructors with required parameters [17], and the technology developed by Ko on the whyline [8] and Hundhausen's ALVIS [6] have influenced the design of Quorum's auditory omniscient debugger [13], which is used by sighted, blind, and visually impaired individuals.

3. Language Comparison Study

The preliminary study presented in this work is designed to test novice accuracy rates as they attempt to program using various features in programming languages. In this section, we discuss how this study was put together, including a brief description of our methodology for measuring accuracy, *Artifact Encoding*. As Artifact Encoding is complex and a thorough description would be too long for this work, readers interested in both a thorough description and empirical validation of the paradigm should see previous work [15]. Our goal in this study was to test the following null hypothesis, H_0: *Novices programming a computer for the first time* *will have equal accuracy rates, regardless of the programming language used.* The study was conducted as a repeated-measures between-subjects design with six tasks. Participants completed all six tasks using the same programming language, in one of the following three groups:

- **Quorum**: A programming language designed using data in empirical studies on the "intuitiveness [14]" of program constructs. The name of this language was previously Hop. Full documentation of the syntax and semantics is available online at `http://sourceforge.net/apps/trac/sodbeans/wiki/Hop`.
- **Perl**: A well-known commercial programming language.
- **Randomo**: A programming language based largely on the syntactical structure of Quorum. With the exception of braces, the lexical rule for variable names, and a few operators (e.g., addition, subtraction, multiplication, division), many of the keywords and symbols were chosen randomly from the ASCII table.

3.1 Participants

We solicited 19 participants between the months of April and July of 2011 from non-computer programming classes at Southern Illinois University Edwardsville after appropriate Institutional Review Board ethics reviews. Participants in all groups were equally paid ten dollars for their participation. All participants reported in an exit survey that they had never before attempted to program a computer. Of these individuals, one participant left in the middle of the study. Since this participant did not complete the experiment, that individual's data was subsequently removed. Of the remaining 18 participants, the average age was 21.3 years, with 12 males and 6 females.

3.2 Procedure and Experimental Walkthrough

When participants began the study, they were first greeted by a proctor and assigned randomly to a group. Once assigned, participants were seated at a computer with Windows 7 installed. Cardboard barriers were placed in between each individual. These barriers prevented people from looking at other participants' screens or otherwise cheating. For a participant to cheat, they would have had to physically stand up and walk to a neighbor's computer to do so. As we monitored participants and made reasonable restrictions to prevent cheating, we conclude that each participant worked independently and without assistance from their peers or the Internet.

Each experimental session lasted approximately two hours and followed a standard checklist of procedures. A complete replication package, including all tasks, procedure checklists, task solutions, and scripts is available from the authors on request. Before beginning, participants were read a script describing what they will be doing in the experiment. Once complete, participants were given a code sample worksheet for the particular language group they were in (Quorum, Perl, or Randomo). See Figure 1 for one of the examples on the reference sheet given to participants.

Task	Procedure and tested concepts
Task 1	7 Minutes, reference: *conditional statements, strings, and variables*
Task 2	7 Minutes, reference: *loops, variables*
Task 3	7 Minutes, reference: *functions, parameters, return values*
Task 4	10 Minutes, no reference: *loops, variables*
Task 5	10 Minutes, no reference: *functions, parameters*
Task 6	10 Minutes, no reference: *nested conditional statements, variables*

Table 1. A table giving the programming concepts highlighted in each task. The amount of time available to complete a task and whether a reference sheet (during) and solution (after) were available for inspection is listed.

The general idea of the experiment is to give novice users code samples similarly to if a participant was learning to program from home on the Internet. As such, we did not train participants on what each line of syntax actually did in the computer programs. Instead, participants attempted to derive the meaning of the computer code on their own.

Participants completed a total of six experimental tasks (see Table 1). In the first three, participants were allowed to reference and use the code samples shown in Figure 1. Seven minutes were allotted to complete each of the first three tasks. Once time was up, participants were given an answer key. This somewhat mimics the idea of finding a working example on the Internet then trying to adapt it for another purpose. For the final three tasks, use of the code samples was not allowed, no solutions were given, and ten minutes was allotted for each task.

3.3 Materials and Tasks

For each experimental task, participants were given identical English descriptions of the code they were asked to write (Quorum, Perl, or Randomo). For task 1, the description read: *Using the code sample given to you, try to write code that defines a variable x that stores real values and is set to 175.3. The code should also define a variable y that stores a string of characters and saves the word false in it. The code should then check whether x is larger than 100. If so, y should save the word true. Otherwise, y should save the words still false. Write your code in the text editor open on the PC in front of you.* See Figure 2 for one possible solution in each programming language. The other five tasks were similar and the concepts participants had to attempt to program are summarized in Table 1.

3.4 Results

To grade each experimental task, we used a simplified version of Artifact Encoding [15]. While previous work discusses the procedure in more detail, Artifact Encoding essentially produces a key that can be graded by a computer. These answer keys are metaphorically similar to how an answer key for a class would be constructed, break-down the computer code into components (e.g., did the user define a particular variable correctly?) and score each individual

```
action Main
 number x = z(1, 100, 3)
end

action z(integer a, integer b,
  integer c) returns number
 number d = 0.0
 number e = 0.0
 integer i = a
 repeat b - a times
  if i mod c = 0 then
   d = d + 1
  end
  else then
   e = e + 1
  end
  i = i + 1
 end
 if d > e then
  return d
 end
 else then
  return e
 end
end
```

(a) Quorum

```
$x = &z(1, 100, 3);

sub z{
 $a = $_[0];
 $b = $_[1];
 $c = $_[2];
 $d = 0.0;
 $e = 0.0;
 for ($i = $a; $i <= $b; $i++){
  if ($i % $c == 0) {
   $d = $d + 1;
  }
  else {
   $e = $e + 1;
  }
 }
 if ($d > $e) {
  $d;
 }
 else {
  $e;
 }
}
```

(b) Perl

```
^ Main {
  ~ x \ z(1, 100, 3)
}

^ z(@ a % @ b % @ c) | ~ {
  ~ d \ 0.0
  ~ e \ 0.0
  @ i \ a
  # (b - a) {
   : i ; c ! 0 {
    d \ d + 1
   }
   , {
    e \ e + 1
   }
   i \ i + 1
  }
  : d ` e {
   _ d
  }
  , {
   _ e
  }
}
```

(c) Randomo

Figure 1. This code shows one of the code samples provided to participants. The description said the following: *This code will count the number of values that are and are not divisible by c and lie between a and b. It then compares the number of values that are and are not divisible by c and makes the greater of them available to the user.*

```
number x = 175.3
text y = "false"
if x > 100 then
 y = "true"
end
else then
 y = "still false"
end
```

(a) Quorum

```
$x = 175.3;
$y = 'false';
if ($x > 100) {
 $y = 'true';
}
else {
 $y = 'still false';
}
```

(b) Perl

```
~ x \ 175.3
?? y \ ?false?
: x ` 100 {
 y \ ?true?
}
, {
 y \ ?still false?
}
```

(c) Randomo

Figure 2. For each programming language, this code represents one possible solution a participant might give for task 1.

piece with a code, in this case 0 or 1. In this study, if an answer was correct, we marked that component with a 1. If a particular component was incorrect, we marked it with a 0. Once all components were marked, a total and a percentage was computed for each task. So, in effect, we computed a "percent correct" metric for each task and used these values in our statistical models, but we did so in such a way that we could compute an inter-rater reliability analysis. Other forms of artifact encoding are far more complex [15].

To ensure our grading could be replicated by other researchers, we first verified that independent raters of the data would give approximately the same result. We did this using a standard inter-rater reliability test called a Kappa analysis (see e.g., Hubert [5]). Two researchers first trained on data not used in the study and then independently coded approximately 20% of the actual data. A Kappa statistic of 0.80 (raw agreement 91.0%) was found, a result which is typically interpreted as highly reliable. Researchers trained in the grading technique proceeded to code the remaining 80% of the data.

We then proceeded to analyze the data using a standard repeated-measures ANOVA test, with corresponding post-hoc Tukey tests and partial-eta squared values using the statistical package SPSS. For those unfamiliar with this procedure, one must first verify that the assumptions of the statistical test are not violated. To do so, we first ran Mauchly's test for sphericity, $\chi^2(14) = 12.071, p = .608$. As the result was non-significant, it implies that the sphericity assumption has not been violated. As such, the standard Greenhouse-Geisser correction was unnecessary for our data.

Next, we conducted a test for within-subjects effects to see if learning played a role in our experiment. Results show that total cross-language averages for task 1 (M=.412, SD=.237) raised slightly by the end of the experiment (M=.552, SD=.224), $F(5, 75) = 3.22, p = .011, \eta_p^2 = .177$). This is not surprising. We would expect participants to improve slightly as they become familiar with either the language or protocol, although it is interesting that scores continued to rise despite the lack of a reference sheet in tasks 4-6. The learning effect interaction with language was non-significant, $F(10, 75) = .735, p = .689, \eta_p^2 = .089$,

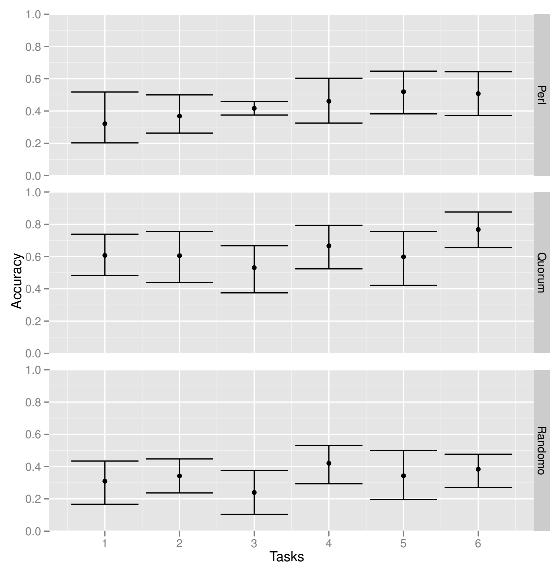

Figure 3. A summary of the accuracy scores by language and task. 1.0 means 100% correct, whereas 0.0 means 0% correct.

implying there was no obvious interaction between task order and language. A summary of the accuracy data for each task is shown in Figure 3.

However, the test most critical to our hypothesis is the test of between-subjects effects—did differences in the languages themselves matter? This test shows that differences between the programming languages were both significant and *very large*, $F(2,15) = 7.759, p = .005, \eta_p^2 = .508$. Further, post-hoc Tukey HSD tests of the between-subjects effect indicate a surprising result. While users of Quorum (Average M=.628, SD=.198) were able to program statistically significantly more accurately than users of Perl (Average M=.432, SD=.179), $p = .047$, and users of Randomo (Average M=.341, SD=.173), $p = .004$, Perl users were *not* able to program significantly more accurately than Randomo, $p = .458$.

3.5 Discussion

But why was it that Quorum, just another general-purpose, object-oriented, programming language afforded better accuracy while programming for novices? We think the most plausible explanation is that we were *very careful* to use empirical evidence for the design of our language. In previous work, we carefully studied what we called "intuitiveness" metrics for a large selection of word choices in the language. As an example, consider the use of the word `for`, as is commonly used to represent the concept of looping in a programming language. The word `repeat` is rated by novices as being nearly seven times more intuitive [14].

Further, the between-subjects effect size (η_p^2=.508) was very large. This statistic is commonly interpreted as the percent of the variance accounted for, in this case evidence that programming language differences might account for up to *half of the problems novices initially face*.

Perl users in our study performed notably poorly, not only performing less well than Quorum, but no better than a language designed largely by chance. While Perl has never had a particular reputation for clarity, the fact that our data shows that there is only a 55.2 % (1 - p) chance that Perl affords more accurate performance amongst novices than Randomo, a language that even we, as the designers, find excruciatingly difficult to understand, was very surprising. This is especially true, we think, considering we chose to test only the syntax in Perl that is relatively common across a number of languages (e.g., `if` statements, loops, functions, parameters). Considering that Java syntax, which many would arguably consider to be easier to understand than Perl, uses similar syntax, we are curious how it would perform. Given this interesting first result, we plan to test a number of additional languages using the same procedures.

3.6 Threats to Validity

A number of potential threats may have influenced the results presented here. First, the sample size in our experiment was low: 18 people spread across 3 groups. It is possible that we just happened to get six high performing novices in the Quorum (or Randomo or Perl) group or six low performing individuals similarly. While the statistical probability of having good or bad individuals is equal across languages, we think that replication on a larger sample of individuals is warranted.

Second, while we have made claims about particular programming languages, it could very well be the case that, for either Randomo or Perl, we did not test a feature of Randomo or Perl in which novices would have performed well. While this is true, we generally tried to test the features that would be common to learn in an introductory computer science course. Other features should be tested, but the ones we chose seemed to be a reasonable first attempt.

We should also mention that our test was conducted with students that had no programming experience. We conducted a test in this way largely because students in our classroom often exhibit substantial difficulty when first learning to program. Students tell us that the syntax they are learning (in C++ at our school), makes no sense and some have difficulty writing even basic computer programs. It is unclear from the work here exactly what kind of students or professionals such results *could* generalize to. For example, there is some evidence that sensible API designs [18] even benefit those with experience. The exact generalizability is unclear, but the results here seem reasonable under the conditions of our test and it is plausible that they extend farther.

4. Summary and Future Work

We have shown in this work an empirical study comparing three computer programming languages: Quorum, Perl, and

Randomo. We compared novices that were programming for the first time using each of these languages, testing how accurately they could write simple programs using common program constructs. Results showed Quorum afforded significantly greater accuracy amongst novices compared to Perl and Randomo, while Perl users were unable to write programs more accurately than those using a language designed with syntax chosen randomly from the ASCII table.

Acknowledgments

We would like to thank the National Science Foundation under grant no. (CNS-0940521) for their generous support of this work. Additionally, we would like to thank Jeff Wilson and Ryan Braun for their help in conducting the experiments.

References

[1] M. H. Brown and J. Hershberger. Colour and sound in algorithm animation. In *Proceedings of the IEEE Workshop on Visual Languages*, pages 10–17, Los Alamitos, CA, 1991. IEEE Computer Society Press. doi: 10.1109/WVL.1991.238856.

[2] R. Garlick and E. C. Cankaya. Using Alice in CS1: A quantitative experiment. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, ITiCSE '10, pages 165–168, New York, NY, 2010. ACM. ISBN 978-1-60558-820-9. doi: http://doi.acm.org/10.1145/1822090.1822138.

[3] S. Hanenberg. Faith, hope, and love: an essay on software science's neglect of human factors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 933–946, New York, NY, 2010. ACM. ISBN 978-1-4503-0203-6. doi: http://doi.acm.org/10.1145/1869459.1869536.

[4] R. C. Holt and J. R. Cordy. The Turing programming language. *Communications of the ACM*, 31:1410–1423, December 1988. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/53580.53581.

[5] L. Hubert. Kappa revisited. *Psychological Bulletin*, 84(2): 289–297, 1977.

[6] C. D. Hundhausen, S. F. Farley, and J. L. Brown. Can direct manipulation lower the barriers to computer programming and promote transfer of training?: An experimental study. *ACM Transactions on Computer Human Interaction*, 16(3):1–40, 2009. ISSN 1073-0516. doi: http://doi.acm.org/10.1145/1592440.1592442.

[7] C. Kelleher and R. Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys*, 37(2):83–137, 2005. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/1089733.1089734.

[8] A. J. Ko and B. A. Myers. Finding causes of program output with the java whyline. In *CHI '09: Proceedings of the 27th International Conference on Human Factors in Computing Systems*, pages 1569–1578, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-246-7. doi: http://doi.acm.org/10.1145/1518701.1518942.

[9] S. Markstrum. Staking claims: a history of programming language design claims and evidence: A positional work in progress. In *Evaluation and Usability of Programming Languages and Tools*, PLATEAU '10, pages 7:1–7:5, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0547-1. doi: http://doi.acm.org/10.1145/1937117.1937124.

[10] L. K. McIver. *Syntactic and Semantic Issues in Introductory Programming Education*. PhD thesis, Monash University, 2001.

[11] B. A. Myers, A. J. Ko, S. Y. Park, J. Stylos, T. D. LaToza, and J. Beaton. More natural end-user software engineering. In *WEUSE '08: Proceedings of the 4th International Workshop on End-User Software Engineering*, pages 30–34, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-034-0. doi: http://doi.acm.org/10.1145/1370847.1370854.

[12] N. Pennington. Comprehension strategies in programming. In G. M. Olson, S. Sheppard, E. Soloway, and B. Shneiderman, editors, *Empirical Studies of Programmers: Second Workshop*, pages 100–113, Westport, CT, 1987. Greenwood Publishing Group Inc.

[13] G. Pothier, E. Tanter, and J. Piquer. Scalable omniscient debugging. In *OOPSLA '07: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 535–552, New York, NY, 2007. ACM. ISBN 978-1-59593-786-5. doi: http://doi.acm.org/10.1145/1297027.1297067.

[14] A. Stefik and E. Gellenbeck. Empirical studies on programming language stimuli. *Software Quality Journal*, 19(1):65–99, 2011. ISSN 0963-9314. doi: http://dx.doi.org/10.1007/s11219-010-9106-7.

[15] A. Stefik, C. Hundhausen, and R. Patterson. An empirical investigation into the design of auditory cues to enhance computer program comprehension. *International Journal of Human-Computer Studies*, 69(12):820 – 838, 2011. ISSN 1071-5819. doi: 10.1016/j.ijhcs.2011.07.002. URL `http://www.sciencedirect.com/science/article/pii/S1071581911000814`.

[16] A. Stefik, C. Hundhausen, and D. Smith. On the design of an educational infrastructure for the blind and visually impaired in computer science. In *Proceedings of The 42nd ACM Technical Symposium on Computer Science Education*, New York, NY, 2011. ACM.

[17] J. Stylos and S. Clarke. Usability implications of requiring parameters in objects' constructors. *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 529–539, May 2007. ISSN 0270-5257. doi: 10.1109/ICSE.2007.92.

[18] J. Stylos and B. A. Myers. The implications of method placement on API learnability. In *SIGSOFT '08/FSE-16: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 105–112, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-995-1. doi: http://doi.acm.org/10.1145/1453101.1453117.

[19] J. Taylor. Analysing novices analysing prolog: what stories do novices tell themselves about prolog? *Instructional Science*, 19:283–309, 1990. ISSN 0020-4277. doi: http://dx.doi.org/10.1007/BF00116442.