An Empirical Comparison of Seven Programming Languages

Often heated, debates about different programming languages remain inconclusive. The author takes a first step toward providing hard data about the relative effectiveness of C, C++, Java, Perl, Python, Rexx, and Tcl.

Lutz Prechelt
University of Karlsruhe

When it comes to the pros and cons of various programming languages, programmers and computer scientists alike usually hold strong opinions. By comparing several languages, I seek to provide some objective information about C, C++, Java, Perl, Python, Rexx, and Tcl.

For the comparison, I used the same program, which implements the same set of requirements for each language. Doing so makes the comparison narrow but homogeneous. Further, for each language, I analyze several separate implementations by different programmers. Such a groupwise comparison offers two advantages. First, it smoothes out the differences among individual programmers, which could threaten the validity of any comparison based on a single implementation per language. Second, it allows us to assess and compare the variability of the program properties the different languages induce.

The comparison investigates several aspects of each language, including program length, programming effort, runtime efficiency, memory consumption, and reliability. I also consider the languages both individually and combined into groups. S*cripting* languages such as Perl, Python, Rexx, and Tcl tend more toward being interpreted than compiled, at least during the program development phase, and they typically do not require variable declarations.

The more *conventional* programming languages—C, C++, and Java—are compiled rather than interpreted, and they require typed variable declarations. Since Java is often believed to be very inefficient, I also sometimes consider C and C++ as one group and Java as another.

PLOTS AND STATISTICAL METHODS

My study used the multiple boxplot display shown in Figure 1 for its main evaluation tool. Each of the lines represents one subset of data, with its name appearing to the left. Each small circle stands for an individual data value. The rest of the plot provides visual aids for the comparison of two or more such data subsets. The shaded box indicates the range of the middle half of the data, that is, from the first (25 percent) quartile to the third (75 percent) quartile. The "whiskers" to the left and right of the box indicate the data's bottom and top 10 percent, respectively. The fat dot within the box is the median (50 percent) quartile. The "M" and the dashed line around it indicate the arithmetic mean and the mean's plus-and-minus-one standard error.

For quantitatively describing the variability within one group of values, I use the bad-to-good ratio: Imagine the data split into an upper and lower half, with the bad-to-good ratio being the median of the upper half divided by the median of the lower half. In the boxplot, the median is the value at the right edge of the box divided by the value at the left edge. In contrast to a variability measure such as the standard deviation, the bad-to-good ratio is robust against outliers.

Most significant observations can easily be made directly from the plots. To be sure, however, I also performed statistical tests. Medians are compared using the one-sided Mann-Whitney U-test (also known as the Wilcoxon rank sum test). The result of each test is a p-the value, that is, a probability that the observed differences between the samples are only accidental and that either no difference or a difference in the opposite direction between the underlying populations

0018-9162/00/$10.00 © 2000 IEEE

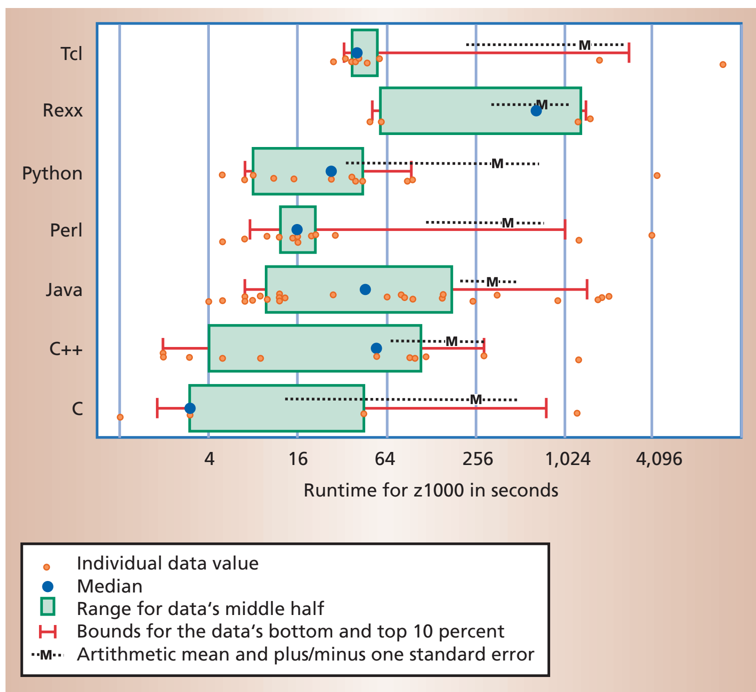

Figure 1. Program runtime on the z1000 data set. Three programs timed out with no output after about 21 minutes. The bad-to-good ratios range from 1.5 for Tcl up to 27 for C++. Note the logarithmic axis. The legend shown here applies to Figures 2 through 7 as well. Refer to the "Plots and Statistical Methods" section for more details.

does indeed exist. I will usually not give the p-value itself, but rather say "... is larger than ..." if $0 < p \leq 0.10$ or "... tends to be larger than ..." if $0.10 < p \leq 0.20$. If $p > 0.10$, there is "no significant difference."

At several points I also provide confidence intervals, either on the differences in means or on the differences in logarithms of means—that is, on the ratios of means. The confidence levels are chosen so that they are open ended, with their upper end at infinity. I computed the confidence intervals using bootstrapping, which is described in more detail elsewhere.[1]

Given the study's validity caveats, these quantitative statistical inference results merely indicate trends and should not be considered precise evidence.

Table 1 shows the number of programs considered for each language and the execution platforms used. The Java evaluation uses either the JDK 1.2.2 Hotspot Reference version or the JDK 1.2.1 Solaris Production version with JIT, depending on which one was faster for each program. I executed all programs on a 300-MHz Sun Ultra II workstation with 256 Mbytes of RAM, running under SunOS 5.7. The results for C and Rexx are based on only five and four programs, respectively, and are thus rather coarse estimates of reality, but for all other languages, the results derive from 10 or more programs, which is a broad-enough base for reasonably precise results.

Table 1. Programming-language comparison statistics.

Language	Number of programs	Compiler or execution platform
Tcl	10	Tcl 8.2.2
Rexx	4	Regina 0.08g
Python	13	Python 1.5.2
Perl	13	Perl 5.005_02
Java	24	Sun JDK 1.2.1/1.2.2
C++	11	GNU g++ 2.7.2
C	5	GNU gcc 2.7.2

The "The Programming Problem: Phonecode" and "Comparison Validity" sidebars describe the study's setup and validity. Readers interested in a more detailed description of the study can find it on the Web at ftp.ira.uka.de.[2]

RESULTS

I evaluated the programs using three different input files: z1000, which contains 1,000 nonempty random

Comparison Validity

Any programming language comparison based on actual sample programs is valid only to the degree to which the capabilities of the respective programmers using these languages are similar. In our case, we only need the programs to be comparable on average, not in individual cases. Several factors present program comparability threats for the 80 programs analyzed in the study.

The programs come from two different sources. The Java, C, and C++ programs were produced in 1997 and 1998 during a controlled experiment in which all the subjects were computer science master students.[1] The Perl, Python, Rexx, and Tcl programs were produced under more variable conditions, created by volunteers after I posted a "call for programs" message on several newsgroups. Consequently, these subjects show more diversity in background and experience.

Programmer capabilities

It is plausible that a public call for programs may attract only fairly competent programmers, hence the script programs reflect higher average programmer capabilities than the nonscript programs. However, two observations suggest that this discrepancy is not a problem. First, with some exceptions, the students who created the nonscript programs were also quite capable and experienced. Second, a fair fraction of the script programmers described themselves as either beginners in programming with their respective scripting language or as not having a thorough programming background, including a VLSI designer, a system administrator, and a social scientist.

Within the nonscript group, the Java programmers tended to be less experienced in their language than the C and C++ programmers because Java was still a new language in 1997 and 1998. In the script group, the Perl subjects may have been more capable than the others because the Perl language appears to attract especially capable programmers—at least that is my personal impression.

Work time reporting accuracy

In contrast to the nonscript programs

The Programming Problem: Phonecode

All programs in the study implement the same functionality, namely a conversion from telephone numbers into word strings, as follows.

The program first loads a dictionary of 73,113 words into memory from a 938-Kbyte flat-text file that consists of one word per line. It then reads "telephone numbers" from another file, converts them one by one into word sequences, and prints the results. A fixed mapping of characters to digits defines the conversion as follows:

```
e   jnq  rwx  dsy  ft   am   civ  bku   lop   ghz
0   111  222  333  44   55   666  777   888   999
```

This means for instance the digit 5 may map into either an a or an m. The program must find a sequence of words such that the sequence of characters in these words exactly corresponds to the sequence of digits in the phone number. All possible solutions must be found and printed. The program creates the solutions word-by-word and, if no word from the dictionary can be inserted at some point during that process, a single digit from the phone number can appear in the result at that position. Many phone numbers have no solution at all. Here is an example of the program output for the phone number "3586-75," where the dictionary contained the words "Dali," "um," "Sao," "da," "Pik," and 73,108 others:

```
3586-75: Dali um
3586-75: Sao 6 um
3586-75: da Pik 5
```

A list of partial solutions must be maintained by the program while it processes each number, and the dictionary must be embedded in a supporting data structure, such as a 10-ary digit tree, for efficient access.

phone numbers; m1000, which contains 1,000 arbitrary random phone numbers with empty records allowed; and z0, which contains no phone numbers at all and thus serves to measure dictionary load time only.

Runtime

I started my analysis by investigating the total runtime, then examined the initialization and search phases separately.

Total: z1000 data set. Figure 1 shows that, except for C++, Java, and Rexx, the programs run in less than one minute. This data suggests several meaningful comparisons.

- Tcl's median runtime is not significantly longer than that for Java or even C++.
- Both Python's and Perl's median run times are shorter than those of Rexx and Tcl.
- C++'s median can be confusing. Because the distance to the next larger and smaller points is rather large, it is unstable. The Wilcoxon test, which takes the whole sample into account, confirms that C++'s median time tends to be shorter than the Java median, with $p = 0.18$.
- C's median runtime is shorter than Java's, Rexx's, and Tcl's and tends to be shorter than Perl's and Python's.
- Tcl's and Perl's runtimes—except for two very slow programs—tend to exhibit less variability than do other languages' runtimes.

We should avoid overinterpreting the plots for C and Rexx, which have only a few points. The Rexx programs can be made to run about four times faster by recompiling the Regina interpreter to use a larger hash table size; the additional memory overhead is negligible. If we aggregate the languages into only three groups (one with C and C++, one with Java, and one with scripts), C and C++ are faster than Java ($p = 0.074$) and tend to be faster than scripts ($p = 0.15$).

from the controlled experiment, for which we know the real programming time accurately, nothing kept the script programmers from "rounding down" the working times they reported when they submitted their programs. Worse, some apparently read the requirements days before they actually began implementing the solution. One such programmer reported reading the program two weeks before the start of coding, "… during which my subconscious may have already worked on the solution."

Evidence, however, indicates that the script group's average work times are reasonably accurate, too: Common software engineering wisdom, which says that "the number of lines written per hour is independent of the language," holds fairly well across all languages. Even better, the same data also confirms that the programmer capabilities are not higher in the script group.

Different task and work conditions

The nonscript group's instructions focused on correctness as the main goal; acceptance tests required high reliability and at least some efficiency. The script group's instructions mentioned eight other program quality goals besides the main goal of correctness. In place of the acceptance test in the nonscript group, the script group received the z1000 input and output data for their own testing. Both these differences may represent an inherent script group advantage.

Summary

Overall, given the design of the data collection, the script group's data will likely reflect several relevant if modest a priori advantages over the nonscript group's data. It is also likely that some modest differences exist in the average programmer's capability between any two of the languages. These threats to validity suggest that we should discount small differences among any of the languages, as they might be based on data weaknesses. Large differences, however, are likely to be valid.

Reference

1. L. Prechelt and B. Unger, *A Controlled Experiment on the Effects of PSP Training: Detailed Description and Evaluation*, Tech. Report 1/1999, Fakultät für Informatik, Universität Karlsruhe, Germany, Mar. 1999, ftp.ira.uka.de.

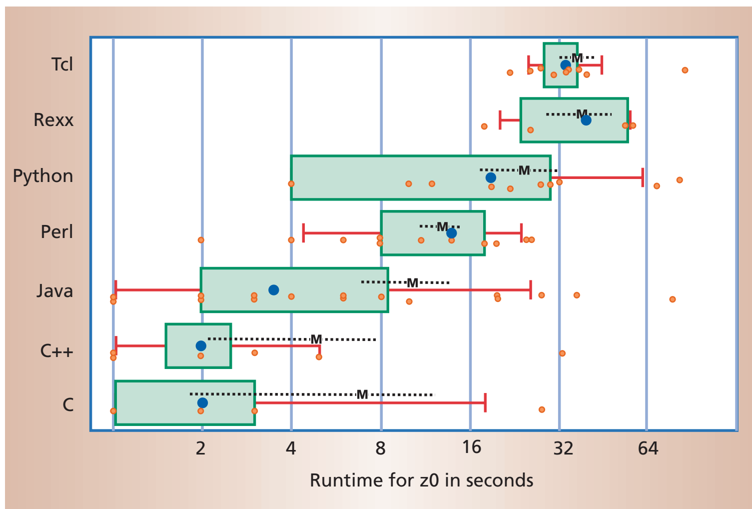

Figure 2. Program runtime for loading and preprocessing the dictionary only (z0 data set). Note the logarithmic axis. The bad-to-good ratios range from 1.3 for Tcl up to 7.5 for Python.

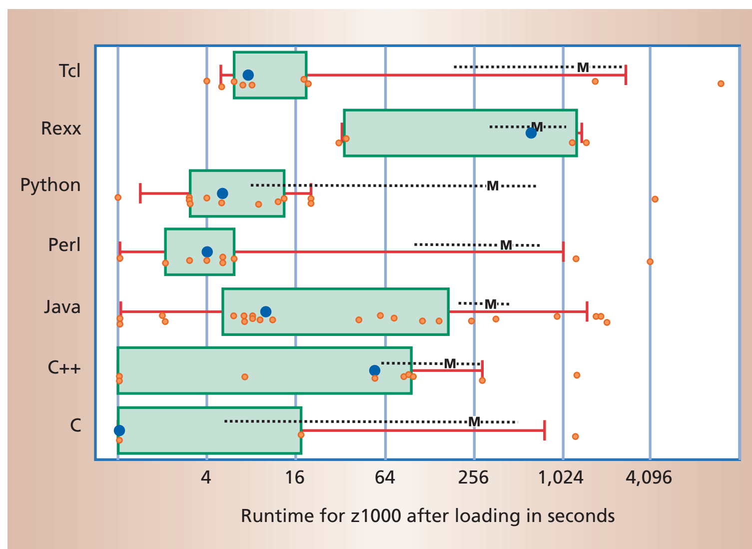

Figure 3. Program runtime for the search phase only, computed as time for the z1000 data set minus time for the z0 data set. Note the logarithmic axis. The bad-to-good ratios range from 2.9 for Perl up to more than 50 for C++.

There is no significant difference between the average Java and script runtimes. The results give an 80 percent confidence that a script will run at least 1.29 times as long—and a Java program at least 1.22 times as long—as a C or C++ program. The bad-to-good ratios are much smaller for scripts (4.1), than for Java (18) or even C and C++ (35).

Initialization phase only, z0 data set. I next focused on the time to read, preprocess, and store the dictionary. Figure 2 shows the corresponding runtimes. The results clearly show C and C++ to be faster in this phase than all other languages tested. The fastest script languages are again Perl and Python.

For the aggregate grouping, we find with an 80 percent confidence level that, compared to a C or C++ program, a Java program will run at least 1.3 times as long, while a script will run at least 5.5 times as long. Compared to a Java program, a script will run at least 3.2 times as long.

Search phase only. Finally, I subtracted the runtime for the loading phase (z0 data set) from the total runtime (z1000 data set) to obtain the runtime for the actual search phase only. Figure 3 shows the corresponding runtimes, which reveal the following:

- Very fast programs occur in all languages except for Rexx and Tcl, while very slow programs occur in all languages.
- Tcl's median run time is longer than the run times for Python, Perl, and C, but shorter than the run time for Rexx.
- Python's median runtime is shorter than the run times for Rexx and Tcl and tends to be shorter than the run time for Java ($p = 0.13$).
- Perl's median run time is shorter than the medians for Rexx, Tcl, and Java.
- C++'s median differs significantly from any other language's.

The group-aggregated comparison indicates no significant differences among any of the groups. However, the results give an 80 percent confidence that the scripts' runtime variability is smaller than that of Java by a factor of at least 2.1, and smaller than that of C and C++ by a factor of at least 3.4.

Memory consumption

Figure 4, which shows the total process size at the end-of-program execution for the z1000 input file, encourages several observations:

- Clearly, the most memory-efficient programs come from the C and C++ groups, while the least memory-efficient programs come from the Java group.
- Except for Tcl, few of the script languages consume more memory than the worst half of the C and C++ programs.
- Tcl scripts require more memory than other scripts.
- For Python and Perl, the relative variability in memory consumption tends to be much smaller than for C and, in particular, C++.
- A select few of the scripts consume large amounts of memory.
- On the average, for the group-aggregated view and with a confidence of 80 percent, the Java programs consume at least 32 Mbytes more memory (297 percent) than the C and C++ programs, and at least 20 Mbytes more memory (98 percent) than the script programs. The script programs consume at least 9 Mbytes more memory (85 percent) than the C and C++ programs.

I conclude from these findings that Java's memory consumption typically runs twice as high as for scripts and that scripts are not necessarily less memory-efficient than a program written in C or C++, although they cannot beat a parsimonious C or C++ program.

Common wisdom suggests algorithmic programs demand a time and memory trade-off: Making a program faster usually requires more memory. Within our given set of programs, this rule holds for all three nonscript languages, but the opposite rule tends to hold for script languages: Scripts that use more memory actually tend to be slower than scripts that use less memory.

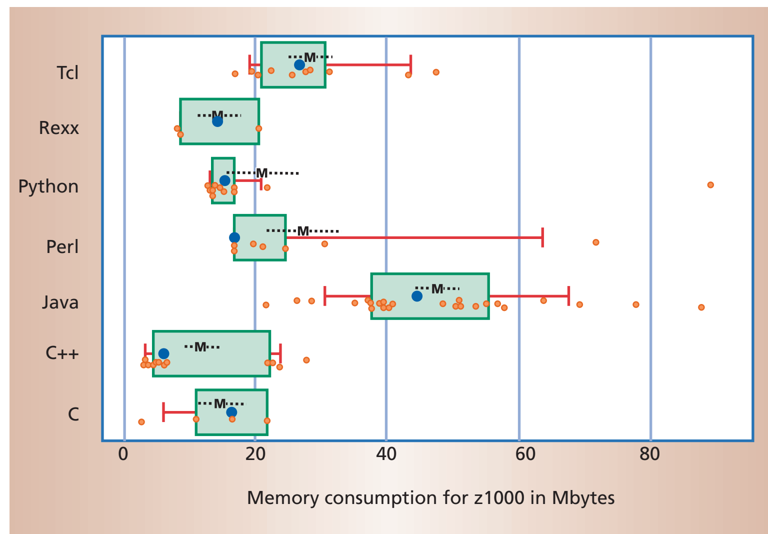

Figure 4. Amount of memory the program required, including interpreter or runtime system, the program itself, and all static and dynamic data structures. The bad-to-good ratios range from 1.2 for Python up to 4.9 for C++.

Length and amount of commenting

Figure 5 shows the number of lines in each program source file that contain anything that contributes to the program's semantics: a statement, declaration, or at least a delimiter such as a closing brace. Nonscripts typically are two to three times longer than scripts. Even the longest scripts are shorter than the average nonscript. At the same time, scripts tend to contain a significantly higher density of comments ($p = 0.020$): Nonscripts average a median of 22 percent as many comment or commented lines as they have statement lines, while the scripts average 34 percent.

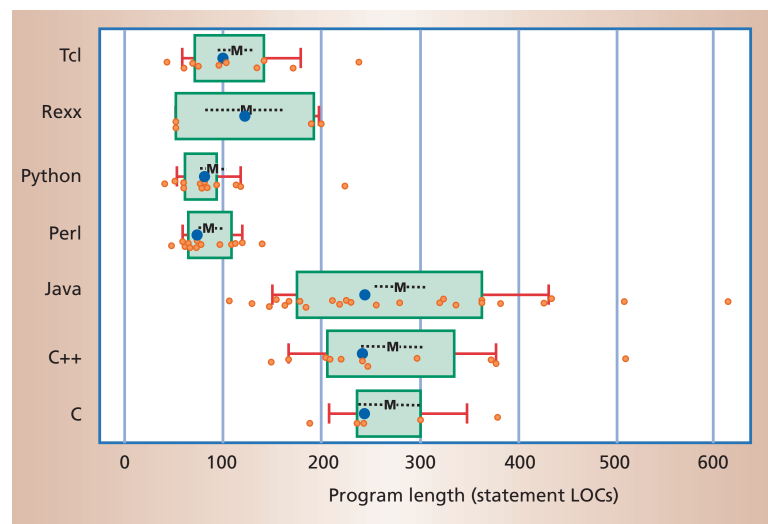

Figure 5. Program length, measured in number of noncomment source lines of code. The bad-to-good ratios range from 1.3 for C up to 2.1 for Java and 3.7 for Rexx.

Program reliability

With the z1000 input file, three programs—one C, one C++, and one Perl—produced no correct outputs at all, either because they could not load the large dictionary or timed out during the load phase. Two Java programs failed with near-zero reliability for other reasons, and one Rexx program produced many of its outputs with incorrect-format-only scripting, resulting in a reliability of 45 percent.

Ignoring these highly faulty programs and comparing the rest by language group—thereby excluding 13 percent of the C and C++ programs, 8 percent of the Java programs, and 5 percent of the script programs—reveals that C and C++ programs are less reliable than the Java and script programs. These differences, however, depend on just a few defective programs and should thus not be generalized.

However, that these differences show the same trend as the fractions of highly faulty programs we're excluding offers good evidence that this reliability ordering among the language groups is real. The scripts' advantages may derive from the better test data available to script programmers, as described in the "Comparison Validity" sidebar.

Next, I compared the behavior for the input file m1000, which allows for phone numbers that do not contain digits, only dashes and slashes. Such a phone number should result in an empty encoding, but we don't usually think of such inputs when reading the requirements. Hence, the m1000 input file tests program robustness. Most programs cope with this situation well, but half of the Java programs and four script programs—one written in Tcl and three in Python—crashed when they encountered the first empty phone number after processing 10 percent of the outputs. Usually, an illegal string subscript or array subscript caused the crash. Of the other programs, 15—one written in C, five in C++, four in Java, two in Perl, two in Python, and one in Rexx—fail exactly on the three empty phone numbers, but work correctly otherwise, resulting in a reliability of 98.4 percent.

Summing up, it appears that scripts are no less reliable than nonscripts.

Work time and productivity

Figure 6 shows the total work time for designing, writing, and testing the program as reported by the

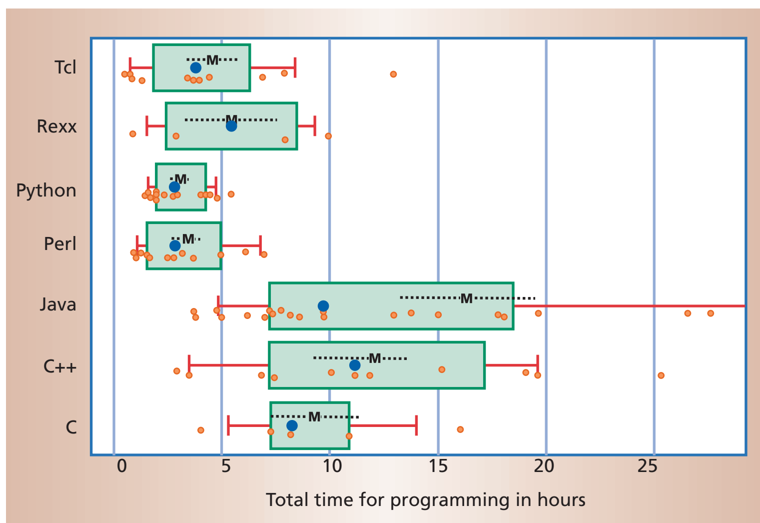

Figure 6. Total working time for realizing the program. The programmers measured and reported the script group times; the experimenter measured the nonscript group times. The bad-to-good ratios range from 1.5 for C up to 3.2 for Perl. Three Java work times of 40, 49, and 63 hours exceed the chart's bounds and thus are not shown.

script programmers and as measured for the nonscript programmers.

We see that scripts, which have a total median of 3.1 hours, take less than half as long to write as nonscripts, which have a total median of 10.0 hours, although validity threats may have exaggerated this difference.

Fortunately, we can check two things at once, namely, the correctness of the work-time reporting and the equivalence of the programmer capabilities in the script versus the nonscript group. Both these possible problems, if present, tend to bias the script group's worktimes downward: We would expect cheaters to fake their time to be shorter, not longer, and, if there is a difference, we expect to see more capable programmers in the script group compared to the nonscript group because in 1997 and 1998, when the study took place, the Java programmers were less experienced relative to the other programmers. This check relies on an old rule of thumb, which says that programmer productivity measured in lines of code per hour is roughly independent of the programming language. Several widely used effort estimation methods—including Barry Boehm's Cocomo[3] and Capers Jones' programming language table for function point estimation[4]—explicitly assume that productivity LOC per hour is independent of programming language. Figure 7 plots the validation of my work-time data, based on this rule. Judging from the reliably known productivity range of Java, all data points, except perhaps the top three Tcl and topmost Perl results, are quite plausible.

None of the median differences show clear statistical significance, although Java versus C, Perl, Python, or Tcl—where $0.07 \leq p \leq 0.10$—comes close to doing so. Even in the group-aggregated view, with its much larger groups, the difference between C and C++ versus scripts is insignificant ($p = 0.22$), and only Java is less productive than scripts ($p = 0.031$), the difference being at least 5.2 LOC per hour, with 80 percent confidence.

This comparison lends credibility to the study's work-time comparison. The times reported for script programming are likely either only modestly optimistic or not at all so. Thus, a work-time advantage for the script languages of about factor two holds. The Java work times appear to be a bit pessimistic because when the study took place, the Java programmers were less experienced than the other programmers.

Program structure

If we consider the designs chosen by those who wrote programs in the various languages studied, we uncover a striking difference. Most programmers in the script group used the associative arrays provided by their language and stored the dictionary words to be retrieved by their number encodings. The search algorithm simply attempts to retrieve from this array, using prefixes of increasing length based on the rest of the remaining current phone number as the key. Any match found leads to a new partial solution to be completed later.

In contrast, essentially all the nonscript programmers chose one of the following solutions: In the simple case, they stored the whole dictionary in an array, usually in both its original character form and the corresponding phone number representation. They then selected and tested one-tenth of the whole dictionary for each digit of the phone number they were encoding, using only the first digit as a key to constrain the search space. This procedure leads to a simple but inefficient solution.

The more elaborate case uses a 10-ary tree in which each node represents a certain digit, nodes at height n representing the nth character of a word. A word is stored at a node if the path from the root to this node represents the word's number encoding. This solution is most efficient, but requires a comparatively large number of statements to implement the tree construction and traversal. In Java, the large resulting number of objects also leads to high memory consumption due to the severe memory overhead incurred per object by current Java implementations.

The script programs require fewer statements relative to the nonscript programs because they do most of the actual search with the hashing algorithm, which is used internally by the associative arrays. In contrast, the nonscript programs require the programmer to code most of the search process's elementary steps explicitly. This difference is further accentuated by the effort—or lack of it—for data structure and variable declarations.

Despite the existence of hash table implementations in both the Java and C++ class libraries, none of the nonscript programmers used them, choosing instead to implement a tree solution by hand. Conversely, the script group's programmers found the hash tables built into their languages to be an obvious choice.

SIGNIFICANT FINDINGS

Comparative analysis of 80 implementations of the phonecode program in seven different languages resulted in the following significant findings:

- Designing and writing the program in Perl, Python, Rexx, or Tcl takes no more than half as much time as writing it in C, C++, or Java—and the resulting program is only half as long.
- I observed no clear differences in program reliability among the language groups.
- The typical script program consumes about twice as much memory as does a C or C++ program. Java programs consume three or four times as much memory as C or C++ programs.
- For the phonecode program's initialization phase, which consists of reading the 1-Mbyte dictionary file and creating the 70K-entry internal data structure, the C and C++ programs complete about three to four times faster than Java programs and about five to 10 times faster than the script languages.
- For the main phase of the phonecode program, which involves searching through the internal data structure, the C and C++ programs run only about twice as fast as Java. The script programs also tend to be faster than the Java programs.
- Within the script languages, Python and Perl run faster than Tcl for both phases.
- For all program aspects investigated, the performance variability that derives from differences among programmers of the same language—as described by the bad-to-good ratios—is on average as large or larger than the variability found among the different languages.

Considering the large number of implementations and the broad range of programmers investigated, this study's results, when taken with a grain of salt, are probably reliable despite the validity threats I've noted. The results, however, can be considered valid only for the phonecode problem; generalizing to different application domains would be risky. For example, I doubt that the relative results for the script languages group would hold up well when applied to other problems.

Despite these caveats, directly comparing different programming languages can provide meaningful insights. For example, I conclude from the study that Java's memory overhead is still huge compared to C or C++, but its runtime efficiency has become quite acceptable. The scripting languages, however, offer reasonable alternatives to C and C++, even for tasks that must handle fair amounts of computation and data. Their relative runtime and memory-consumption overhead will often be acceptable, and they may offer significant advantages with respect to programmer productivity, at least for small programs like the phonecode problem.

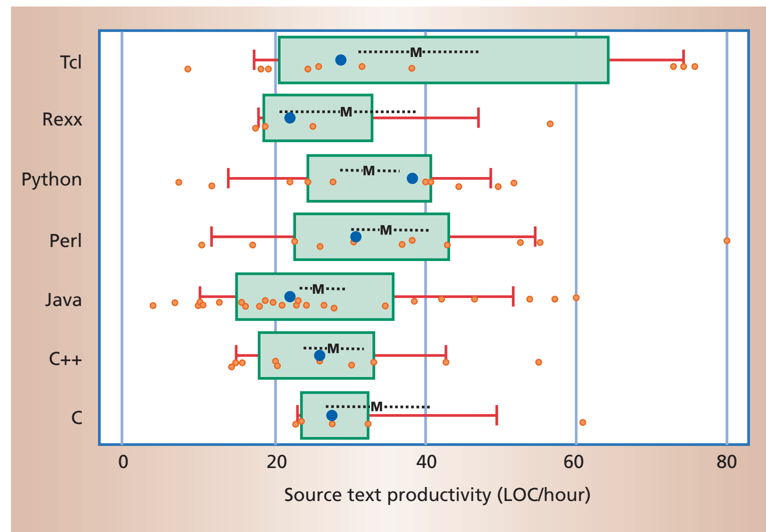

Figure 7. Source text productivity in noncomment lines of code per total work hour. The bad-to-good ratios range from 1.4 for C to 3.1 for Tcl.

I advocate that more and larger studies be conducted along the lines of the one described here. Such work is necessary if we are to disperse the fog of vendor hype and programmer advocacy so that we can see each language's strengths, weaknesses, and peculiarities clearly. Acquiring such knowledge will help advance the state of software development overall. ❖

References

1. E. Bradley and R. Tibshirani, "An Introduction to the Bootstrap," *Monographs on Statistics and Applied Probability 57*, Chapman and Hall, New York, 1993.
2. L. Prechelt, *An Empirical Comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a Search/String-Processing Program*, Tech. Report 2000-5, Fakultät für Informatik, Universität Karlsruhe, Germany, Mar. 2000, ftp.ira.uka.de.
3. B.W. Boehm, *Software Engineering Economics*, Prentice Hall, Englewood Cliffs, N.J., 1981.
4. C. Jones, *Software Productivity Research*, *Programming Languages Table, Version 7*, http://www.spr.com/library/0langtbl.htm, 1996 (as of Feb. 2000).

***Lutz Prechelt** is head of Quality Assurance at abaXX Technology, Stuttgart, Germany. At the time of this study, he was assistant professor at the University of Karlsruhe. His research interests are empirical software engineering, benchmarking issues, parallel processing, and research methodology. Prechelt received a PhD in informatics from the University of Karlsruhe. He is a member of the IEEE Computer Society, the ACM, and the GI. Contact him at lutz@prechelt.de.*