1987-07-01

Can Programmers Reuse Software?

Scott N. Woodfield
woodfield@cs.byu.edu

David W. Embley

Del T. Scott

Original Publication Citation

Woodfield, S. N., D. W. Embley, and D. T. Scott. "Can Programmers Reuse Software?" Software, IEEE 4.4 (1987): 52-9.

Can Programmers Reuse Software?

An experiment asked programmers untrained in reuse to evaluate component reusability. They did poorly. Are reusability's promises hollow? Or are there some answers?

Scott N. Woodfield, David W. Embley, and Del T. Scott
Brigham Young University

Software reusability has long held out the unrealized promise of increased productivity. Researchers have predicted the development of certified software components available for easy incorporation in new systems, and they have presupposed that by the 1990s many software engineers would be more like computer designers determining gross system structure and connections and relying heavily on prefabricated software components.[1] Indeed, where software reuse has been encouraged and practiced, managers report satisfaction and success,[2] and as much as a 35- to 85-percent improvement in productivity can be realized.[3]

Despite the promise and flurry of activity, software is rarely reused in practice. Tracz identifies and discusses several technical, organizational, political, and psychological barriers to software reuse.[4] He observed that the barriers are not insurmountable, but to surmount them

- useful and certified components must be made available,
- tools and methods to support both creation and use of component libraries must be developed, and
- money, time, and personnel must be expended to develop software reuse systems and train users to become proficient with them.

0740-7459/87/0700/0052/$01.00 ©1987 IEEE

From the perspective of a software production staff, profitability depends largely on component availability and on easily locating components, assessing their applicability, and incorporating them into the software system being developed. Because we can do little initially about component availability, and because we consider component incorporation an important issue — but one that is mute unless relevant components can be found and their applicability properly assessed[4,5] — we focus on these issues. Locating software components and assessing their applicability are both based on the same issue: They both compare a specification of what is required with a candidate component in a software-reuse library.

To better understand the process of comparing specified and required components, we conducted an experiment to test if programmers could accurately assess component reusability. We gave 51 software developers 21 treatments in which they were asked whether they would reuse an existing abstract data type to meet an abstract-data-type specification.

The results show that there is considerable confusion. Software-development personnel untrained in reuse cannot assess the worth of reusing a candidate abstract data type to satisfy the implementation requirements of a specified abstract data type. Their decisions were also influenced by some unimportant features — and were *not* influenced by other important ones.

Focus and hypotheses

We restricted our investigation to libraries whose software components are abstract data types and assumed an environment where programmers do object-oriented design and development. We chose to investigate abstract-data-type libraries rather than function libraries because we believe that more resources can be saved per unit resource invested when searching for and reusing an abstract data type than when searching for and reusing a single function.[5,6]

Researchers exploring an unknown area in computer science often rely exclusively on introspection when trying to study issues and answer important questions about human-computer interaction. Unfortunately, as Moran said, a "designer, relying on an egocentric, 'folk psychology' has no way to gauge his intuitions; and intuitions about complex psychological behavior (even about one's own behavior) can be remarkably deceptive."[7]

To avoid these pitfalls, we should gather at least some empirical evidence to help guide the design. Although a single experiment cannot provide all the information needed, much can be gleaned from a well-designed experiment, especially if the experiment is conducted early in the design stage when, because of folk psychology, it is easy to start building a system modeled on invalid assumptions.

We investigated abstract-data-type libraries rather than function libraries because abstract data types can save more resources per unit.

In our experiment we posed four questions:

1. Given an abstract-data-type requirements specification, do subjects properly assess the worth of reusing a candidate abstract data type?

Because the people tested were untrained in software reuse and were given no guidelines or assistance in assessing the worth of reusing a component, we hypothesized that they would be unable to appropriately compare a specification with a candidate software component. We also hypothesized that they would be inconsistent among themselves in their assessments. The responses to this question should establish the degree of need for education and software tools.

2. What features best explain how subjects assess the worth of reusing a candidate abstract data type for a given abstract-data-type requirements specification?

Features we expected to be of interest were missing domain specifications and operations, extraneous domain specifications and operations, implied modifications, and size. If the subjects overlooked important features, that would show that users must be trained to consider these overlooked features and tools must be designed to bring these features to the users' attention. If the subjects based decisions on questionable features, that would show what features users should be warned to ignore.

3. Based on performance and demographics, do distinguishable groups of subjects exist?

Because the subjects all have about the same expertise in software design and development, we hypothesized that no distinguishable groups should exist. If distinguishable groups did exist, we must determine whether different sets of instructional material and software tools should be developed to meet the differing needs.

4. Can a threshold value be predicted that discriminates between acceptance and rejection of a candidate abstract data type for a given abstract-data-type requirements specification?

We hypothesized that a threshold value exists and that training, experience, and the availability and use of software tools may alter the threshold value. A threshold and the knowledge of its value, along with a proper understanding of the important features, would give us a starting point for developing a system that could automatically assess the worth of reusing a candidate component for a specification. Automatic assessment would give us a way to measure algorithmically how closely a candidate component matches a specification, thus giving us an initial means of automatically finding candidate components in a library.

The experiment

To test these hypotheses, we conducted an exploratory experiment that simulated an environment where someone knowledgeable about abstract data types worked in a software design and development project that required abstract data types. We assumed that if time and effort could be saved, the person would rather reuse an abstract data type than create one from scratch. Thus, the basic measurement in the experiment was the person's perception of time saved. By varying the circum-

stances under which a person might decide to reuse a candidate abstract data type and by measuring how much time a person thought might be saved, we hoped to answer our questions.

The subjects. Fifty-one people participated in the experiment. Twenty-five were from three local industries (Hercules, Word Perfect, and Novell). The other 26 were senior and graduate students taking various software engineering classes at Brigham Young University. All the subjects were familiar with abstract data types and had created programs using the concept.

Although the subjects were not picked randomly from the general population of knowledgeable abstract-data-type users, they are representative of both industrial and academic environments and have a wide range of practical experience.

SPECIFICATION 9
room addition

Components of room addition:
height
length
width
.
.
.

Operations on room addition:
COMPUTE_VOLUME_OF(room) -> volume
calculate the total air space of the room
ESTIMATE_CARPET_COSTS(room,price_per_yard) -> dollars
estimate the cost of carpet for the room given the cost per yard

DATA ABTRACTION 9
room addition

Components of room addition:
height
length
width
.
.
.

Operations on room addition:
COMPUTE_VOLUME_OF(room) -> volume
calculate the total air space of the room
ESTIMATE_AIR_CONDITIONING_NEEDS(room,climate)
-> air_conditioning_capacity
estimate the number of tons of air conditioning needed to cool room in summer
.
.
.

Would you reuse this data abstraction to meet the given specification? (circle one) YES NO

If YES, what percent of the estimated creation time do you think you will spend modifying the code (circle one)

0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%

Figure 1. Sample treatment.

Experimental design. We designed an exploratory experiment to investigate the influence of four factors on subjects' perceptions of time saved:

- the percent of the specified abstract data type's domain definition that must be added to the candidate abstract data type,
- the percent of the candidate abstract data type's domain definition that could be deleted,
- the percent of the specified abstract data type's operations that must be added, and
- the percent of the candidate abstract data type's operations that could be deleted.

In an exploratory experiment the number of levels should be minimized, and their values should be chosen to create a distribution of responses about each factor level. The results of a pilot study indicated that there would be a good distribution of results at about the 20-percent and 50-percent addition/deletion levels.

We designed 16 treatments to cover the four factors; each factor had two levels. To make our experiment more representative of reality, we added four treatment combinations. These treatments held the addition/deletion levels for domains at zero and provided combinations of the addition/deletion levels for the two operation factors. To investigate nonlinearity, we added a center point. Thus, the experiment contained 21 treatments (2^4+4+1).

Our study shared a problem common to experimental designs in social science and clinical studies: the need to control subjects' differences. To minimize this problem, each person should take all the treatments. Because subjects' responses may be affected by treatment order, each subject received an instrument containing a different randomized sequence of treatments.

Instrument. Figure 1 shows part of treatment 9. It consists of a specified abstract data type (Specification 9 in the figure) and a candidate abstract data type (Data Abstraction 9), both given in the same format. The first part of both the specification and data abstraction was a

brief domain definition containing a list of components of the abstract data type's domain. The second part of both contained a list of operations defined at a very high level by giving only input and output parameters and brief English descriptions. In all but a few cases each instance fit on one page.

As the figure shows, the subjects were asked two questions for each treatment. First, they were asked if they would reuse the candidate abstract data type to create an abstract data type for the specification. Second, if they answered yes, they were asked to estimate the amount of time it would take to modify the candidate abstract data type to meet the specified abstract data type. This response was to be a percent of the estimated time it would take to create the specified abstract data type from scratch.

Measures. The experiment yielded two types of measures: the subject's responses and quantitative measures on the instrument itself. With 51 subjects, 21 treatments, and two questions, there were 2142 data values for responses.

To fill out the subject-response table completely, a 100-percent value was automatically recorded for all negative responses. The 100-percent value for a negative response means that if an abstract data type is written from scratch, it requires all (100 percent) of the creation time to produce the abstract data type.

A 100-percent response for yes is also possible and means that the person believes it would take the same amount of time either way and that the person chose to reuse the candidate abstract data type.

The person may also believe that it might actually take longer to create the specified abstract data type by reusing the candidate abstract data type, but in this case the response should simply be no. We did not request percents for negative responses because we thought that this might confuse many of the subjects and because it is not clear how much above 100 percent people might estimate.

There are several quantitative measures on the instrument itself. In addition to the designed-in percentage of additions and deletions, the amount of modification necessary is also important — but it is easily overlooked. In general, changes to the domain constituents often imply changes to operations. Although we did not design specific amounts of modification into the instrument, we could measure the amount of modification induced.

We were also interested in the absolute number of operators to be added. We thought that people might consider deletions and domain additions to be insignificant because these changes are relatively simple compared to adding a new operation.

We also thought that the treatment size might influence people. The treatment size varied from one domain constituent to 10 and from four operations to 32.

Effort estimation has long been a problem in software engineering. Most managers use lines-of-code estimates. We used software-science estimates because our projects were small.

Because one of our major objectives was to determine whether people could properly assess the worth of reusing a candidate abstract data type, we needed an estimate of the creation time percentage required to reuse each candidate abstract data type to produce the specified abstract data type. The cost of having each person write the necessary code for all abstract data types in the experiment was estimated to be more than 100 hours per person (much too expensive).

Effort estimation has long been a problem in software engineering. When required, software-production managers have relied most often on lines-of-code estimates.[8] Software science is another technique that can be used.[9] Although software-science estimations appear to be unreliable when applied to large projects, empirical evidence shows that for relatively small functions — like those operators in the treatments of our experiment — this technique is acceptable.

The lines-of-code technique is based on an estimation formula of the form $time = a(\mathrm{LOC})^b$. Basili cites references showing that for several different studies b is close to one and that for small functions it can be taken to be one.[8] We counted lines of code for each operation to be added or modified after having written it in pseudocode.

The software-science technique assumes that the time it would take to implement the function can be estimated from a function specification and knowledge of the implementation language. The formula is $time = ((V^*)^3/\lambda^2)/S$, where λ is a language-level constant, S is the Stroud constant, and $V^* = (2+\eta_2^*)\log_2(2+\eta_2^*)$ is the potential volume and depends only on η_2^*, which is the number of input/output operands for the operation. Once the specification for an operation is given, the number of input/output operands is easily obtained.

Both the software-science estimations and the lines-of-code estimations are sensitive to their multiplicative constants, λ, S, and a. For our analysis, these constants do not affect the results because the constants cancel out when the ratio of estimated efforts is computed.

We obtained the lines-of-code and software-science metrics for our treatments by computing an abstract-data-type modification effort with one of the metrics and dividing by the estimated creation effort derived with the same metric. There were two types of abstract-data-type modification efforts calculated. One was based on the effort needed to create the new operations to be added to an abstract data type, and the other (and more sophisticated) effort estimator also included the effort needed to modify operations. For our purposes the effort needed to modify an operation was assumed to be equal to the creation effort raised to the ⅝ power.[10]

We used both lines-of-code and software-science metrics because one or both may be imperfect. To determine if the two measures were consistent, we computed the means, standard deviations, correlation coefficient, and coefficient of determination for all treatments. The software-science mean and standard deviation are 46.27 and 29.06. The lines-of-code mean and standard deviation are 46.59 and 23.89. The correlation coeffi-

cient is 0.91 and the correlation of determination is 0.83. We also considered the differences between pairs of lines-of-code and software-science estimators and found that the mean difference is −0.32 percent and that the standard deviation of the differences is 12.41.

Statistically, we cannot show that the means or variances are different. From this evidence and from personal evaluation, we are confident that the estimators, when used together, reasonably predict how much effort is required to modify an abstract data type for reuse.

Results

After testing our subjects and determining our metrics, we found the following results.

Worth assessment. Our first working hypothesis was that effort estimations for modifying a candidate abstract data type to satisfy the requirements of a specified abstract data type would be close to the actual effort needed to modify the candidate abstract data type. Because determining the actual amount of effort was beyond our capability to measure accurately, we decided to estimate the actual modification effort associated with each treatment by the average of the software-science and lines-of-code measures.

Our hypothesis thus became that effort estimations for modifying a candidate abstract data type to satisfy the requirements of a specified abstract data type would be close to the software-science/lines-of-code estimate.

One method of measuring closeness is to compare the mean of an effort estimation with the expected value derived from software science/lines of code. Although this let us determine if a person is close to the average, it indicated nothing about whether we could expect a person to be close for a given situation. For example, a person might give wild estimates, but the person's average might still be quite close to the expected value derived from software-science/lines-of-code measures. We were interested not only in whether the average was good but whether every estimate was fairly accurate. Therefore, we wanted not only the mean of the effort estimations to be close to the expected value for each treatment, but we also wanted the variances of the estimates to be small.

Initial observations indicated that the people underestimated the amount of effort needed to modify a candidate abstract data type for reuse and that their estimates varied highly. On average, the people were 15.35 percent low and had a standard deviation of 28.15.

Believing that people were generally low and highly variant, we chose to test two stronger hypotheses: (1) that at least one person's effort estimations over the 21 treatments is consistently close to the software-science/lines-of-code estimates and (2) that there exists at least one question for which the 51 people consistently give an effort estimation close to the software-science/lines-of-code estimate.

Although size should not have been a significant factor, we found that it influenced people unnecessarily.

To test the two hypothesis, we used a goodness-of-fit test based on the chi-squared distribution,[6] χ^2. We let the α level be 0.001, which means we will accept a factor as being significant only if there is a one-in-a-thousand chance we could be wrong. We chose this α level, instead of the usual 0.05, because of the study's exploratory nature — we wanted to identify only the large factors of practical importance.

We can reject the first hypothesis at $\alpha = 0.001$ ($X^2 = 127 > \chi^2_{(0.999,\ 20)} = 45.3$) and conclude that the best person's estimates are not consistently close to the software-science/lines-of-code estimate. These results imply that the best person (and therefore all people) could not accurately estimate the effort needed to modify a candidate abstract data type for reuse.

We can reject the second hypothesis $\alpha = 0.001$ ($X^2 = 454 > \chi^2_{0.999,\ 50)} = 86.7$) and conclude that the estimates were not consistently close to the software-science/lines-of-code estimate for the best question. These results imply that for the best question (and therefore for all questions) the people could not make accurate estimates.

Significant factors. We applied analysis of variance, covariance analysis, and regression techniques to determine which factors best explain the results. The factors of interest were the effects of person variability, answering yes or no, question variability, treatment size as measured in characters and entities, and different types of effort estimation. We analyzed these factors by constructing different models and determining if the factors of the model are statistically and practically significant.

The basic model includes factors for person variability, for decisions about whether to reuse the candidate abstract data type, and for question variability. The first two factors, person variability and yes/no decisions, are nuisance factors, factors of little interest that account for a lot of the overall variability. If ignored, they give a false representation of the unexplained variability. Because each factor explains a certain percent of remaining variance, their order is important, and because we must account for any correlation that the nuisance factors have with the primary factors, the nuisance factors should be put in the model first.

For the basic model, the question variability is the most important factor. We did not consider percent of operation addition and deletion, percent of subdomain addition and deletion, and other correlated values individually because the question-variability factor contains all this information. Thus, we have an overall test to determine whether any of these individual factors or combinations of these factors is important.

The statistics here are based on an analysis of variance of the data. While not perfect, probability plots of the residuals make us confident of the results produced by the analysis of variance. Table 1 shows the results for the basic model. This model accounts for 74.4 percent of the variance. Because the F value for the questions factor (18.86) is greater than the F statistic (2.27), we rejected the hypothesis that all the questions are identical. Hence, some of the individual factors — percent of operation addition and deletion, percent of

subdomain addition and deletion, and other correlated values — are significant.

We thought the size of a particular treatment might cause people to change their estimates of the abstract-data-type modification effort. But although the experiment was designed so size should not have been significant, we found that people responded differently to treatments that differed only in size.

We analyzed two types of size metrics: entity size and character size. Entity size is the number of subdomains and operations in the specification and candidate abstract data types. Character size is the number of characters in the domain and operation descriptions of the specification and candidate abstract data types.

We analyzed these two size metrics by including them in the model as nuisance factors; they were introduced as the third and fourth factors in the model. Because order is important, we ran two analyses. The first analysis used character size as the third factor and entity size as the fourth factor. In the second run, the positions were reversed. Tables 2 and 3 show the results.

Table 1.
Basic model results, $\alpha = 0.001$.

Factor	% variance explained	F statistic	df	F value
Subjects	12.1	9.44	50,988	1.75
Yes/no	52.6	2050.00	1,998	10.83
Questions	9.7	18.86	20,998	2.27

Table 2.
Effect of size, analysis 1, $\alpha = 0.001$.

Factor	% variance explained	F statistic	df	F value
Subjects	12.1	9.44	50,988	1.75
Yes/no	52.6	2049.95	1,998	10.83
Character size	3.6	35.62	4,998	4.62
Entity size	0.8	8.16	4,998	4.62
Question	5.2	16.89	12,998	2.74

Table 3.
Effect of size, analysis 2, $\alpha = 0.001$.

Factor	% variance explained	F statistic	df	F value
Subjects	12.1	9.44	50,988	1.75
Yes/no	52.6	2049.95	1,998	10.83
Entity size	4.3	42.35	4,998	4.62
Character size	0.1	1.29	4,998	4.62
Question	5.2	16.89	12,998	2.74

These results show that, regardless of order, the number of entities in a treatment is significant. Character size, when introduced after entity size, is not significant, even at the 0.1 α level ($F = 1.96$). This implies that entity size can explain character size, but character size cannot account for all the variance due to entity size. We therefore concluded that size is a significant factor and can be represented by the number of entities in a treatment.

We then investigated how people measured effort to determine what factors are best correlated with people's effort estimations. We had three hypotheses:

1. People used the absolute number of subdomains and operations that must be added, deleted, or modified to estimate effort. We call this the absolute-entities measure.
2. People measured the percent of subdomains and operations that needed to be added, deleted, or modified. We call this the relative-entities measure.
3. People estimated the true modification effort in hours and divided by the estimated creation effort. We call this the true-effort-estimation measure. This last measure is our lines-of-code/software-science estimate.

To evaluate these three measures, we made three analyses. Each analysis added one of the measures as the fourth factor to the model (after subjects, yes/no, and entity size); the other two effort measures were not included. The questions factor was retained and became the fifth factor, letting us determine if the new factor explained some of the question variability.

Table 4 shows the results. Unlike the previous tables, Table 4 summarizes the three analyses. Each row in the table shows only the results for the fourth factor in each of the five-factor models. These results show that all three measures are significant but that the relative-entities measure is best. Further analysis showed that the relative-entities measure accounts for most of the variability of the absolute-entities and true-effort-estimation measures.

We believe that people evaluated the effort to modify an abstract data type primarily as the ratio of affected entities to total specified entities. Unfortunately, the true-effort-estimation — which we believe they should have used — seemed to be the least important.

Having concluded that the relative-entities measure is important, we then decomposed it into its five components: subdomain addition, subdomain deletion, operation addition, operation deletion, and operation modification. These are the factors we originally wanted to investigate.

We examined different orderings of the factors in the model and observed that they all lead to the same conclusions. Table 5 presents the results for one of the orderings. These results show that the most influential factor was the percent of operations that must be added to a candidate abstract data type to make it meet a specification. All other factors are unimportant.

Although we expected the addition and deletion of subdomains to be overlooked, we had hoped that the percent of operations to be modified would have been significant.

Distinguishable groups. To determine if subject groupings affected performance, we analyzed the demographics using clus-

Table 4.
Comparison of three effort estimators, $\alpha = 0.001$.

Factor	% total variance explained	% question variance explained	F statistic	fd	F value
Absolute entities	6.3	65.0	61.32	4,998	4.62
Relative entities	7.3	75.5	71.22	4,998	4.62
True effort estimation	4.9	50.6	95.36	2,998	6.91

Table 5.
Analysis of relative-entities measure, $\alpha = 0.001$.

Factor	% total variance explained	% question variance explained	F statistic	fd	F value
Subject	12.10	—	9.44	50,998	1.75
Yes/no	52.60	—	2049.95	1,998	10.83
Entity size	4.30	—	42.35	4,998	4.62
Subdomains added	0.02	0.40	0.90	1,998	10.83
Subdomains deleted	0.00	confounded			
Operations added	4.00	75.50	156.88	1,998	10.83
Operations deleted	0.05	0.99	2.05	1,998	10.83
Operations modified	0.001	0.02	0.06	1,998	10.83
Questions	1.20	23.10	3.99	12,998	2.74

ter analysis. Statistically, there were two (perhaps three) groups.[6] One group contained four people who had significantly more work experience, but these four did not behave measurably differently than the other 47 people. Except for this group, we were unable to give a rational explanation for any other cluster.

Threshold value. Finally, we investigated the relationship between the estimated effort to modify a candidate abstract data type for reuse and the decision to reuse the abstract data type. We were interested in a threshold effort-estimation value that could be used to determine if a person would reuse a candidate abstract data type.

Logic indicates that the modification effort should be less than or equal to the effort needed to create the specified abstract data type. Because the modification-estimation effort was measured as a percent of the creation effort, the threshold value should be 100 percent or less.

Because of the experimental design, we could not determine an exact threshold value from unadjusted responses. If we used unadjusted values, about half the responses should be no and half the responses should be yes. Our experiment concentrated on abstract data types likely to be reused. Also, there was no real relation between negative answers and estimated modification effort because all negative responses were explicitly assigned a modification effort of 100 percent. These problems could be overcome if we analyzed the adjusted responses.

Of the 805 yes responses, none had an adjusted-modification-effort estimation greater than 63.5 percent. Also, none of the 265 no responses had an adjusted-modification-effort estimation less than 73.5 percent. From this evidence, we think that the threshold value is about 70 percent for software-developer populations similar to ours.

If the worth of reusing an abstract data type could accurately be assessed, we would expect a person to reuse an abstract data type even if only a few percent of the creation effort could be saved. The 70-percent threshold value suggests that the people made decisions based not only on possible effort saved, but also on an assessment of risk. The large variance in responses suggests that the people were not sure of their effort estimations. To protect themselves from undue risk, it appears that most people will only reuse an abstract data type if they can save more than 30 percent of the effort.

Our experiment supports these conclusions:

- Software-development personnel untrained in software reuse cannot assess the worth of reusing a candidate abstract data type to satisfy the implementation requirements of a specified abstract data type. Assuming that the expected effort determined by our software-science/lines-of-code measure is a reasonable estimate of expected effort, no person accurately estimated the expected effort for all candidate/specified abstract-data-type pairs, and there was no pair for which all people gave an accurate estimate. The data shows that subject responses were highly variable and generally low when compared to software-science/lines-of-code measures.
- Software-development personnel untrained in software reuse are influenced by some unimportant features and are not influenced by some important ones. People were influenced by the size of the candidate abstract data type and by the percent of additional operations required. People were not influenced by the percent of operators to be modified nor by estimates of effort based on software science and lines of code. Size, per se, is unimportant, and percentage of operator addition is less important than percentage of oper-

ator modification and reasonably accurate estimates of actual effort.

• For people similar to our subjects, demographics are not likely to affect performance. In our experiment, no identifiable groups of people performed differently than the other people.

• For people similar to our subjects, the data suggests that if the effort to reuse a candidate abstract data type is perceived to be less than 70 percent of the effort to create an abstract data type from scratch, the candidate abstract data type would be chosen for reuse. The threshold is less than 100 percent, and after adjustment for people who were consistently higher or lower than the average, the data indicates that the threshold is close to 70 percent.

Some directions for future research are clear. Because users cannot properly assess the worth of reusing a candidate abstract data type to satisfy the requirements of a specified abstract data type, we must provide enough help to avert the bad experiences users are certain to have if left solely to their own resources. User assistance can be provided through education and software tools that summarize appropriate information.

The data shows that untrained users tend to consider size an important feature on which to base decisions about reusability. They shouldn't. On the other hand, users fail to estimate modification effort in any form — and, instead of estimating the amount of time to add operations, they only considered the percent of operations to be added. While the latter factor is a crude estimate of effort, we should be able to do better. Users should be taught to ignore unimportant features and to properly evaluate important ones.

Furthermore, an initial set of tools should concentrate on providing accurate information for informed decision-making. As an initial information set, we suggest the number and percent of operations that must be added and modified. Eventually, the tool should be enhanced to help estimate abstract-data-type creation and modification effort. Before building tools for abstract-data-type reusability, however, we will carry out further controlled experiments to get the data necessary to help us proceed with confidence. □

References

1. A.I. Wasserman and S. Gutz, "The Future of Programming," *Comm. ACM*, March 1982, pp. 196-206.
2. R.G. Lanergan and C.A. Grasso, "Software Engineering with Reusable Design and Code," *Trans. Software Engineering*, Sept. 1984, pp. 498-501.
3. T.C. Jones, "Reusability in Programming: A Survey of the State of the Art," *Trans. Software Engineering*, Sept. 1984, pp. 499-493.
4. W.J. Tracz, "Why Reusable Software Isn't," tech. report, Electrical Engineering Dept., Stanford Univ., Stanford, Calif., 1986.
5. D.W. Embley and S.N. Woodfield, "A Knowledge Structure for Reusing Abstract Data Types," *Proc. Ninth Int'l Software Engineering Conf.*, CS Press, Los Alamitos, Calif., 1987, pp. 360-368.
6. D.W. Embley, D.T. Scott, and S.N. Woodfield, "An Exploratory Experiment Directed Toward Unraveling the Problems of Software Reusability," Tech. Report BYU-CS-87-4, Computer Science Dept., Brigham Young Univ., Provo, Utah, 1987.
7. T.P. Moran, "An Applied Psychology of the User," *ACM Computing Surveys*, March 1981, pp. 1-11.
8. V.R. Basili, "Resource Models," in *Tutorial on Models and Metrics for Software Management and Engineering*, V.R. Basili, ed., CS Press, Los Alamitos, Calif., 1980, pp. 4-9.
9. M.H. Halstead, *Elements of Software Science*, North Holland, New York, 1977.
10. S.M. Thebaut, "The Saturation Effect in Large-Scale Software Development: Its Impact and Control," PhD dissertation, Computer Science Dept., Purdue Univ., West Lafayette, Ind., 1983.

Scott N. Woodfield is an associate professor of computer science at Brigham Young University. Before joining Brigham Young, he was faculty member at Arizona State University. His research interests include software reusability and metrics.

Woodfield received a PhD in computer science from Purdue University. He is a member of IEEE and ACM.

David W. Embley is a professor of computer science at Brigham Young University. Before joining Brigham Young, he was a faculty member at the University of Nebraska. His research interests include database query languages, object-oriented systems, and software reusability.

Embley received a PhD in computer science from the University of Illinois. He is a member of ACM.

Del T. Scott is an associate professor of statistics at Brigham Young University. He is also a consultant on the development of the Statistics Dept.'s computer facilities. He has developed statistical programs for linear models.

Scott received a BS and MS from Brigham Young University and a PhD from Pennsylvania State University, all in statistics. He is a member of ASA, the Royal Statistical Society, and the Biometric Society.

Embley and Woodfield can be reached at Computer Science Dept., Brigham Young University, Provo, UT 84602. Scott can be contacted at 244 TMCB, Brigham Young University, Provo, UT 84602.