METRICS FOR TARGETING CANDIDATES FOR REUSE: AN EXPERIMENTAL APPROACH

Stephen R. Schach and Xuefeng Yang

Vanderbilt University

Abstract

Accidental reuse of a software component is the utilization of that component in a new product, where the component was not specifically constructed for the purpose of possible future reuse. Software metrics have been used to try to identify which of the literally billions of available software components would make good candidates for accidental reuse. The selected candidates are then submitted to a domain expert for further study. In this paper, we point out that the decision regarding the reuse of a software code module is made on the basis of more than just the source code, whereas the metrics are computed from only the source code. Furthermore, the deciding factor is what we term "perceived reusability," that is to say, a domain expert decides, on largely subjective grounds, whether or not to reuse a specific module. We conducted a experiment in which we measured various metrics for a number of different code modules. The results of our experiment were that those metrics could not be used to predict perceived reusability. Consequently, we concluded that perceived reusability is largely subjective, and hence that automatic tools should not be used to predict whether a module would make a good candidate for future accidental reuse.

Key Words

Reusability, accidental reuse, software metrics, experimentation.

Introduction

Software reuse refers to the utilization of a software component *C* within a product *P*, where the original motivation for constructing *C* was other than for use in *P*. Component *C* may be any software component including, but not restricted to, a specification, plan, contract, design, or code module, or part of such a component. The idea underlying reuse is that we can produce software faster and more cheaply by reusing existing software components (with or without modification), rather than by building a new product from scratch. More formally, component *C* is *reusable* if the effort required to reuse it is significantly smaller than the effort required to implement a component with the same or similar functionality [12].

There are two types of reuse. *Deliberate reuse* occurs when component *C* was originally constructed for the purpose of possible reuse in future products that were not specified at the time that *C* was constructed, whereas *accidental reuse* occurs when the motivation for constructing *C* was as a part of some product *Q* not identical to *P* [13]. There are distinct advantages to deliberate reuse over accidental reuse.

For example, if a component is specifically constructed with possible future reuse in mind, it is likely that such a component will be more carefully written, and more thoroughly tested and documented, than is usually the case with software components. Also, deliberate reuse components usually share a uniformity of style that makes both reuse and future maintenance easier. However, deliberate reuse has a major drawback in that it is expensive. The cost of specifying, designing, testing, and documenting a software component to the standards required for potential future reuse will certainly not be recouped if that component is never reused. In fact, the cost may not be recouped if the component is reused only a small number of times in future products.

The solution seems clear, namely accidental reuse. There are innumerable software components that have been constructed as part of existing software products. All that has to be done is to extract the relevant components, place them in some sort of reuse library, index them in some way, and then retrieve them when needed. Whereas the problems of indexing and retrieval have largely been solved (for example, see [9, 11]), the real issue is: Which of the literally billions of existing software components should be selected for future reuse?

Research has been conducted on the use of metrics to predict whether an existing software component is a good candidate for future reuse. For example, in the work of Chen and Ramamoorthy [4], Caldiera and Basili [3], and Dunn and Knight [6], the approach taken is to compute various software metrics for a code module. .Then, depending on the values of the metrics, the module is either discarded, or handed to a (human) domain expert for further analysis regarding reusability. That is to say, automatic tools are used to decide whether a given module is a good candidate for further study, thus saving the time of the domain expert.

The problem is that we do not know whether this approach works. That is to say, no research has as yet been published that shows that all the discarded modules are indeed poor candidates for reuse, and that many (or most) of the selected modules are later identified by the domain expert as good candidates for reuse. On the contrary, we believe that the metrics-based approach has two weaknesses. First, an important criterion for reuse is that the component be considered *useful*. No metric has yet been proposed for measuring this automatically, and it is unlikely that this will be achieved in the foreseeable future. For this reason, it is likely that many candidate components labeled as potentially reusable on the basis of metrics may in fact be rejected by a domain expert. Second, the decision as to whether to reuse a component is not made exclusively on the basis of the source code. A software developer will decide whether or not to reuse a particular software component on the basis of all information at his or her disposal. For example, in order to decide whether to reuse a code component, the software developer will examine the source code, comments within the source code, fault reports, reports regarding previous reuse experiences with that component, and any other relevant documentation. Thus, if documentation is indeed available, it will be utilized in a reuse decision. However, there are no automatic tools that can peruse arbitrary documentation and use this in-

formation, together with metrics based on the source code, to judge reusability. On the other hand, if there is no documentation available, then it is unlikely that the module will be reused at all.

The important criterion is whether a software component is judged by a domain expert to be reusable. If not, it will never be reused. In what follows, we use the term "perceived reusability" of a component C to denote that a number of software professionals with relevant domain knowledge have examined component C, and the body of opinion is that C is easy to reuse. That is to say, they have decided that, if in the future they need a component with the functionality of C, it will be significantly easier to use C (either unchanged or modified) than to produce from scratch a new component C' with functionality either identical or similar to that of C.

We have conducted an experiment to determine if software metrics can be used to predict whether or not a given module would be perceived to be reusable. These metrics fell into two classes, namely objective metrics, and perceived metrics. An example of an *objective metric* is cyclomatic complexity [10], essentially, the number of different paths through a code module [5]. There exists a straightforward algorithm for computing the cyclomatic complexity of any code module. This algorithm uses only the source code itself. The cyclomatic complexity will be the same whether or nor there is any documentation of any kind, and whether or not such documentation is complete, correct, and up-to-date. The result of computing the cyclomatic complexity is a number that can then be used in statistical analyses, for example, to determine if it correlates with perceived reusability.

An example of a *perceived metric* is perceived complexity. In order to determine the perceived complexity, the module, together with all available relevant documentation, is shown to a number of software professionals. Each decides whether that module, in their opinion, has a high level of complexity, an average level of complexity, or a low level of complexity. (The word "complexity" is used in its nontechnical sense of "complicated" or "intricate.") If there is some degree of agreement among the participants regarding the perceived complexity of the module, then we have confidence regarding the value of this perceived metric.

We did not expect to find much correlation between an objective metric and the corresponding perceived metric, because the objective metric is a number that is computed from the software component alone. On the other hand, a perceived metric is estimated on the basis of all the information that would be available to the human who will decide whether or not to reuse the software component being considered, including domain information. For the same reason, we also did not anticipate that objective metrics could be used to predict perceived reusability. However, we were interested to see whether there was any correlation between one or more perceived metrics and perceived reusability.

In order to investigate these issues, we chose a number of software components. For each component we computed 8 objective metrics. Subjects then assigned values to 6 perceived metrics, including the perceived reusability of each component. We then analyzed the results.

This paper is organized as follows. The various metrics are defined in the next section. Then we describe the experiment. The data analysis is presented in the following section, and we end the paper with our results and conclusions.

Metrics of This Study

Metrics were selected in two ways. First, we chose metrics that other researchers in the field have investigated from the viewpoint of predicting future reusability. Examples of such metrics included cyclomatic complexity, and number of formal parameters. Others were metrics that we felt might be relevant, such as number of global variables. We considered a total of 14 metrics, 8 of which were objective, and 6 perceived. We first describe the objective metrics.

1. Complexity. Complexity reflects the difficulty in designing, coding, testing, or maintaining a software component. McCabe's *cyclomatic complexity* is defined as the cyclomatic complexity number $v(G)$ of the underlying flowgraph. For a flowgraph G with e edges and n nodes, $v(G)$ is given by

$$v(G) = e - n + 2$$

The computation of $v(G)$ can be greatly simplified by noting that $v(G) = DE + 1$, where DE is the decision count, the number of branches within the component [5].

High cyclomatic complexity is thought to be negatively correlated with reusability [3]. However, low cyclomatic complexity frequently implies low functional usefulness. Thus, we did not anticipate a linear relationship between cyclomatic complexity and perceived reusability.

2. Volume. Volume is a measure of the size of a component. In terms of Halstead's software science indicators [7], volume V is defined to be

$$V = (N_1 + N_2)\log_2(n_1 + n_2)$$

where n_1 is the number of unique operators, n_2 is the number of unique operands, N_1 is the total number of operators, and N_2 is the total number of operands in the component .

Volume could be a characteristic of a reusable component since the reuse value of too small a component might not justify the cost of re-engineering, while too large a component is frequently fault-prone, and thus has potential quality problems [3].

3. Regularity. Regularity is another software science measure, the ratio of the actual length of a component to the predicted length. It is given by the formula [7]

$$r = \frac{n_1 \times \log_2 n_1 + n_2 \times \log_2 n_2}{N_1 + N_2}$$

It has been suggested that a component with regularity close to 1 is readable and nonredundant, and hence reusable [3]. However, we saw no reason to anticipate that regularity would correlate with *perceived* reusability.

4. Number of comment words. Comments are an important, sometimes essential, way of understanding the functionality and structure of a component. Up to a certain point, the more comments, the greater the likelihood that the component will be understood. However, too many comments can reduce the comprehensibility of a component. Thus, we expected the number of comments to be related to perceived reusability, but nonlinearly.

5. Number of numerical constants. It is hard to understand a component with a large number of numerical constants. Furthermore, modifying such a component is difficult. Thus, this metric might be a measure of reusability.

Table I. Objective Metrics for the Seven Selected Modules.

Metric	Module						
	builtin_function	global_conflicts	build_field_call	finish_decl	init_decl_processing	convert_harshness	c_decode_option
Complexity	5	27	17	35	7	90	63
Volume	702	4,183	1,988	4,167	14,410	10,995	4,995
Regularity	1.3	1.1	0.8	0.7	0.7	0.4	0.3
No. of Comment Words	114	274	107	520	373	453	62
No. of Numerical Constants	9	29	4	27	30	84	73
No. of Global Variables	12	27	27	34	166	40	23
No. of Formal Parameters	4	0	5	3	0	3	1
Av. Length of Variable Names	12	10	12	13	16	12	17

6. Number of global variables. This metric appears to have asimilar impact on reusability as number of numerical constants.

7. Number of formal parameters. A component with a large number of formal parameters is generally hard to reuse. Selby has reported that, in an environment in which reuse was permitted but not actively encouraged, reused components could generally be characterized by having a small number of formal parameters [14].

8. Average length of variable names. We had anticipated that perceived reusability would increase with average length of variable names, but would then decrease as the average length continued to increase to the point of unwieldliness.

Now we describe the 6 perceived metrics.

9. Perceived reusability. A domain expert assigns this metric a value from 1 to 5, where 1 denotes "not reusable" (that is, extremely difficult to comprehend and/or does not fulfill a useful purpose), and 5 denotes "reusable" (that is, easy to understand, and can be directly reused in other contexts with little or no modification).

10. Perceived complexity. A domain expert decides whether the component is very complex, complex, or easy, within the context of future reuse.

11. Perceived number of comments. A domain expert decides whether the number of comments is insufficient, enough, or too many, within the context of future reuse.

12. Perceived length of variable names. A domain expert decides whether the lengths of the variable names are too short, appropriate, or too long, within the context of future reuse.

13. Perceived number of global variables. A domain expert decides whether there are too many global variables or if the number of global variables is appropriate, within the context of future reuse.

14. Perceived number of formal parameters. A domain expert decides whether there are too many formal parameters, or if the number of formal parameters is appropriate, within the context of future reuse.

We now describe the experiment we conducted to measure the correlations between these various metrics.

Experimental Details

The experiment was performed in two phases. First, we selected appropriate software modules from a public domain package. Then we asked a group of subjects to evaluate the reusability and the 5 other perceived metrics for each of the modules.

The modules to be evaluated had to be carefully chosen. First, it was important to select components that were not written with reuse in mind; the purpose of our experiment was to study accidental reuse. Second, we wanted to use high-quality software, because poor quality software is almost impossible to reuse. Third, we needed an application domain that was not too specific, so that all the subjects would have some knowledge of that domain. Fourth, the components had to be from the same package to assure a reasonable uniformity of quality and programming style. Finally, the components had to have a broad range of values for the objective metrics in order to be able to detect statistical dependencies.

In order to compute the values of the objective metrics we wrote a static code analyzer. It is amusing that the analyzer consists largely of reused code, including UNIX utilities and a graphical front end taken from a different sort of static analyzer. We used the analyzer to help us select C components (functions) from the GNU C compiler *gcc*. The values of the objective metrics for the seven modules we selected are shown in Table I. As can be seen from that table, the values of the objective metrics vary as widely as we had wished.

The subjects of the experiments were 39 computer science students. The majority were graduate students, but there were a few final-year undergraduates (we do not know the exact number as a consequence of our attempts to maintain confidentiality of responses). It would certainly have been better to have used software professionals [13]. Nevertheless, there are those who feel that the use of students as software engineering subjects is justified [1, 2].

All the subjects were familiar with C. We provided them with an oral explanation of the purpose of the survey, and an outline of reusability. Then, written instructions and a careful definition of reusability were attached to the listings of the seven modules. The subjects were told that they did not have to return the survey, if they chose not to, for any reason. As a result, only 20 of the 39 who had agreed to take part actually returned their surveys. As a result, we feel confident that the surveys

Table II. Results of MANOVA for Perceived Reusability z-Scores.

Module	Mean	Standard Deviation
builtin_function	0.164	0.769
global_conflicts	-0.266	0.821
build_field_call	0.291	0.739
finish_decl	-0.467	0.539
init_decl_processing	-0.001	0.932
convert_harshness	-0.207	0.910
c_decode_option	0.485	0.925
Between-subjects effects: $F = 1.69$ ($P < 0.209$)		
Within-subjects effects: $F_{6, 14} = 2.92$ ($P < 0.011$)		

were thoughtfully filled out by participants who considered themselves qualified to do so.

A separate evaluation form was attached to the listing of each module; all available information about the module was included in the listing. The subjects were asked to review each module, and then fill out the form. The results were then analyzed.

Data Analysis

The first step was to analyze the perceived reusability data in order to ensure that the seven modules were assigned significantly different perceived reusability scores. After all, the seven modules have dramatically different objective metrics and functionalities, so we expected different perceived levels of reusability. Only if this held would there be any point in looking further to find what determined the differences.

We used MANOVA (multivariate analysis of variance) [8] to analyze the perceived reusability scores. We were concerned that the individual variability might adversely influence the analysis of reusability scores, because each subject might have his or her own standard of evaluation. In order to minimize individual variability, it was necessary to transform the reusability scores to z-scores [8]. Consequently we applied MANOVA to the z-scores. The results, as shown in Table II, showed that, in general, the subjects did perceive the reusability of the modules differently. The F-test revealed a reusability difference that was significant ($P < 0.011$). This result was encouraging, and suggested that further analysis was needed to determine what features of the modules affected their perceived reusability.

Table III. Correlation Coefficient Matrix for Perceived Reusability and Objective Metrics.

	CMPLX	VOL	REGU	COMM	NUMC	LNAME	GLOBV	FPAR
PREUS	-0.126	-0.276	-0.108	-0.926	-0.037	-0.446	-0.131	0.115
CMPLX		0.274	-0.759	0.260	0.922	0.097	-0.282	-0.040
VOL			-0.506	0.544	0.497	0.419	0.832	-0.535
REGU				-0.207	-0.794	-0.617	-0.170	0.163
COMM					0.227	-0.147	0.366	-0.174
NUMC						0.356	-0.010	-0.340
LNAME							0.484	-0.350
GLOBV								-0.514

Legend: PREUS: perceived reusability; CMPLX: complexity; VOL: volume; REGU: regularity; COMM: number of comment words; NUMC: number of numerical constants; LNAME: average length of variable names; GLOBV: number of global variables; FPAR: number of formal parameters.

Table IV. Correlation Coefficient Matrix for Perceived Metrics.

	PCMPLX	PCOMM	PLNAME	PGLOBV	PFPAR
PREUS	-0.932	-0.055	-0.583	0.039	-0.819
PCMPLX		0.110	0.491	-0.172	0.623
PCOMM			0.516	-0.098	-0.283
PLNAME				-0.090	0.531
PGLOBV					-0.130

Legend: PREUS: perceived reusability; PCMPLX: perceived complexity; PCOMM: perceived number of comment words; PLNAME: perceived length of variable names; PGLOBV: perceived number of global variables; PFPAR: perceived number of formal parameters.

Specifically, correlation analysis might yield some insights into the relationship between the metrics and perceived reusability.

First we considered the objective metrics. The correlation coefficients are shown in Table III. There is no linear relationship between perceived reusability and all but one of the objective metrics. The one exception is the number of comment words, which is *negatively* correlated (coefficient of correlation -0.926, $P < 0.003$). Superficially, this does not make much sense; it implies that the fewer the comments, the greater the likelihood of perceived reusability. Our interpretation of this result is that reading the comments convinced certain subjects that the modules in question were not reusable. In the absence of comments, there was some doubt in their minds as to reusability, and they decided that it was possible to reuse those modules.

Next, we analyzed the perceived metrics. The results are shown in Table IV. Two of the correlation coefficients are significant, namely the coefficient between perceived reusability and perceived complexity ($P < 0.002$), and between perceived reusability and perceived number of formal parameters ($P < 0.023$). When multiple regression analysis was applied to the data, the corresponding regression coefficients were also significant (see Table V). Thus, only these two perceived metrics can be used to predict perceived reusability.

Finally, we measured the pairwise correlation between each of the 5 corresponding objective and perceived metrics. This is shown in Table VI, which reflects the fact that there is no linear relationship between an objective metric and the corresponding perceived metric. The closest to a linear relationship occurs with the metric number of formal parameters, but even here $P < 0.069$. The objective metric number of formal parameters is simply a count of how many formal parameters there are. The corresponding perceived metric is an indication as to the extent to which a domain expert feels that the number of formal parameters is appropriate within the context of reuse. Thus, the fact that there is no significant linear relationship between the two metrics is not too surprising.

Table V. Multiple Regression Analysis of Perceived Reusability Scores against Perceived Metrics.

Perceived Metric	Regression Coefficient	P-Value (F-test)
Perceived Complexity	-0.030	0.005
Perceived No. of Formal Parameters	-0.101	0.036
(Intercept)	2.889	0.000

Table VI. Pairwise Correlation Analysis of Corresponding Objective and Perceived Metrics.

Metric	Correlation Coefficient	P-Value (F-test)
Complexity	0.031	0.948
Number of Comment Words	-0.267	0.563
Length of Variable Names	-0.035	0.941
No. of Global Variables	-0.331	0.469
No. of Formal Parameters	0.719	0.069

Results and Conclusions

The results we obtained from the statistical analysis of the data generally confirmed what we had anticipated. The major results we obtained were as follows:

1. There was a great variability in the evaluations of the modules by the subjects. For example, 4 of the 20 subjects assigned reusability score 1 to every module, that is, they indicated that they did not believe that any of the modules could be reused for any purpose. There are also significant differences between the evaluations of the modules by the other 16 subjects. The possible reasons for this include:
   (a) Reuse of these modules is genuinely problematic.
   (b) Some of the subjects may have deemed the available documentation to be inadequate for reuse purposes.
   (c) Some of the subjects may have incomplete domain knowledge.

If reason (c) is true, then this suggests that a domain analysis expert system should be incorporated into an automated tool for identifying reusable modules. This would be extremely hard to achieve, and a major obstacle to implementing a program of accidental reuse. If reasons (a) or (b) are true, then this suggests that there are other major obstacles to accidental reuse. Instead, reusable modules must be designed from the beginning with future reuse in mind, that is, planned reuse is the way to achieve reuse.

2. There are significant differences in the perceived reusabilities recorded by the subjects. This implies that perceived reuse is subjective. This, too, suggests that there are major obstacles to accidental reuse.

3. The only linear relationship that was observed between perceived reusability and the 8 objective metrics was that between perceived reusability and number of comment words. However, this was a negative correlation, and thus hard to believe. We conclude that our objective metrics cannot be used to predict reusability. Consequently, we feel that an automated tool based on software metrics would not be reliable.

4. The objective metrics were not found to be linearly related to their perceived counterparts. This result supports our contention that software metrics cannot be employed to predict perceived reusability.

5. On the other hand, the relationships between perceived reusability and *perceived* complexity and *perceived* number of formal parameters were significant ($P < 0.002$ and $P < 0.024$, respectively). This result confirms our view that accidental reusability is subjective. Whereas it indicates that prediction of reusability is possible, the problem is how to measure the perceived metrics of a reusable module automatically. In passing, the result suggests that a development method that reduces perceived complexity would enhance perceived reusability. An example of such a method is the object-oriented paradigm.

Our overall conclusion is that accidental reusability is very much in the eye of the beholder. As a consequence, we do not believe that a tool that scans existing code modules could successfully predict which modules would make good candidates for future reuse.

Address for Correspondence

Stephen R. Schach
Computer Science Department
Vanderbilt University
Box 1679, Station B
Nashville, TN 37235
U.S.A.

E-mail: srs@vuse.vanderbilt.edu

References

1. Boehm-Davis, D. A., Holt, R. W., and Schultz, A. C. The Role of Program Structure in Software Maintenance. *International Journal of Man-Machine Studies* 36 (1992), 21–63.
2. Brooks, R. Studying Programmer Behavior Experimentally: The Problems of Proper Methodology. *Communications of the ACM* 23 (1980), 207–213.
3. Caldiera, G. and Basili, V. R. Identifying and Qualifying Reusable Software Components. *IEEE Computer* 24, 2 (1991), pp. 61–70.
4. Chen, Y. F., and Ramamoorthy, C. V. The C Information Abstractor. Proc. Compsac 86, Chicago, Ill., October 1986.
5. Conte, S. D., Dunsmore, H. E., and Shen, V. Y. *Software Engineering Metrics and Models.* Benjamin/Cummings, Menlo Park, Calif., 1986.
6. Dunn, M. F., and Knight, J. C. Automating the Detection of Reusable Parts in Existing Software. In *Proc. 15th International Conference on Software Engineering*, Baltimore, Md., May 1993, pp. 381–390.
7. Halstead, M. H. *Elements of Software Science.* Elsevier North-Holland, New York, N.Y., 1977.
8. Halstead, D. C. *Statistical Methods for Psychology.* PWS-KENT, Boston, Mass., 1987.
9. Maarek, Y. S., Berry, D. M., and Kaiser, G. E. An Information Retrieval Approach for Automatically Constructing Software Libraries. *IEEE Trans. on Software Engineering* 17 (1991), 800–813.
10. McCabe, T. J. A Complexity Measure. *IEEE Transactions on Software Engineering* SE-2 (1976), 308–320.
11. Prieto-Díaz, R. Implementing Faceted Classification for Software Reuse. *Communications of the ACM* 34 (1991), 88–97.
12. Prieto-Díaz, R. and Freeman, P. Classifying Software for Reusability. *IEEE Software* 4, 1 (1987), 6–16.
13. Schach, S. R. *Classical and Object-Oriented Software Engineering.* Third Edition, Richard D. Irwin, Homewood, Ill., 1995 (to appear).
14. Selby, R. W. Quantitative Studies of Software Reuse. In *Software Reusability. Volume II: Applications and Experience*, T. J. Biggerstaff and A. J. Perlis (Eds), ACM Press, New York, N.Y., pp. 213–233, 1989.