Measuring Software for Its Reuse Potential

James A. Hess; USACECOM; Ft. Monmouth

Key Words: Software, Reuse, Design, Software Metric, Modeling

ABSTRACT

The so-called 'software crisis' has led to a search for more productive methods of software development. Improvements in areas such as software tools, modern programming practices, new computer languages, and personnel capability have been inadequate to meet the needs of today's systems. Software productivity improvements of an order of magnitude or more may be possible through the use of reusable software. The use of a software components catalog from which software parts could be assembled was proposed over twenty years ago. Practical application of these concepts has been limited to a few demonstrations in selected application areas. Some critical software measurement issues must be addressed before software reuse can become a reality. This paper examines a variety of measures for software, shows why the physical properties of software are important for determining reuse potential, and provides design guidelines for the structuring and analysis of software to enhance its reusability.

INTRODUCTION

The amount of money spent on computer software has been growing at an alarming rate. It has been estimated that the annual cost of computer software and services for the US DOD will increase from $6.6 billion in FY80 to $37.2 billion in FY90. Furthermore, if the cost of software continues to increase at this rate, computer software will consume the entire defense budget by the year 2015. Clearly, steps need to be taken to ameliorate this situation. One of the steps has been the search for more productive methods of software development. The term productive, as defined by Webster, means "of or engaged in the creating of economic value, or the producing of goods and services." In this paper, a measure of software productivity is defined as the change in the cost of producing a given software function as new development techniques are introduced. The cost of producing software can be estimated from the number of man-hours expended in its development.

A commonly held misconception is that significant increases in software productivity can be achieved through the introduction of modern programming practices. Boehm maintains: "An effective software productivity improvement program involves much more than introducing modern programming practices" (Ref. 3). Boehm's COCOMO software cost estimating model uses several software cost driver attributes as effort multipliers. These factors 'provide a natural framework for identifying the high leverage productivity improvement factors' and their payoffs. When most of these attributes are varied from worst case to optimum conditions, the software productivity effect of these key factors is relatively small. Experience with a language or in an application area, the use of powerful software tools or modern programming practices, and design requirements for reliability provide less than a 2:1 improvement in productivity.

The only factor examined by Boehm providing significant productivity improvement is the capability of the people involved in the project. He identified a potential impact of over 4:1. Unfortunately, at the same time the requirements for software are growing by an order of magnitude per decade, the shortage in the number of qualified software professionals is also growing. The requirement for software professionals has been growing at approximately 12% per year during the '80's, but the supply of new practitioners has been growing at only 4% per year.

The combination of small productivity increases and the shortage of highly qualified professionals limits the payoff of the factors listed above. Since conventional means of improving the software development process will not yield the required increases in productivity, new approaches must be investigated. One such approach, the reuse of already developed software, offers tremendous potential for increases in productivity. Because the effect of software reuse is to reduce or eliminate the need for new software, there is no limit on the resulting productivity improvement. Productivity increases of over an order of magnitude are feasible.

SOFTWARE COMPONENTS ENGINEERING

Approximately ten years ago, Belady (Ref. 2) provided a simple prescription for fixing the software development process. He observed: "The most striking aspect of contemporary software development is the virtually unlimited freedom of action for everybody involved. The entire team enjoys the same freedom. Everybody is an individual craftsman, constructing large or small components in a so-called hierarchical arrangement. The architect ... probably can't read the code anyway - the language he learned is not used anymore.' Belady goes on to suggest the eighties must result in a better balance between 'free innovation and disciplined evolution.' He foresaw the challenge for the eighties to be the evolution of a means to evolve software from the current large program inventory. He calls for an inventory of easily modifiable and reconfigurable software components and the birth of a real sense of software engineering. Simply stated, software should be developed 'by combining parts selected from among tested alternatives.'

There are several difficult technical and managerial problems to be solved before this vision of software components engineering will be realized. The hardest problem may be finding a common denominator to use in structuring the software components. The factors to investigate include: finding some measure of 'hidden commonality' among software components, and determining optimum size for the components. One of the limitations of today's technology is the inability to capitalize on software components no matter what their size. As the size of the components increases, the simplicity of constructing a system from those components increases thereby increasing their value for reuse. On the other hand, as component size increases the probability of a component being 'just right' decreases and its reuse value decreases. The challenge is to increase the reuse value or payoff for a component of some arbitrary size. Other problems which must be solved include: finding a way to represent the design of software components, describing and selecting components, and educating both the technical and manage-

U.S. Government work not protected by U.S. Copyright.

1988 PROCEEDINGS Annual RELIABILITY AND MAINTAINABILITY Symposium
Componentry Concepts

Over ten years before Belady’s essay, McIlroy (Ref. 15) proposed a similar notion. He felt software people “get the short end of the stick in confrontation with hardware people because they are the industrialists and we are the craftsmen.” McIlroy suggested a parallel between certain industrial techniques and the production of software. The concept of subassemblies carried over directly. The mass production of interchangeable parts corresponded to the software concept of “modularity.”

Finally, machine tools used in industrial production had an analog in the tools software developers use: editors, compilers, and debuggers. The analogy falls apart, however, for other characteristics of mass production. There are no manufacturers of standard parts nor are there catalogs of parts qualified for reuse.

The software industry, McIlroy claimed, is weakly founded. Proof of this weakness is the absence of a software components subindustry. McIlroy’s software componentry concept is based on three steps: development of families of components, selection of components based on measurable characteristics, and synthesis of software based on rational principles. Many families of components must be characterized, developed, and represented with some sort of software ‘blueprint.’ The varieties of components for a given job ought to be broad so users do not pay a penalty in size or speed for unwanted generality. The components must meet some criteria relating to their quality, intelligibility, and tailorable ability. The software engineer’s function is to select appropriate components based on some criteria and then assemble the components to make systems and subsystems using a building block approach.

Technical Issues

The reuse of software components as the basis for a software engineering industry faces considerable obstacles as well as technical problems. Issues relating to profitability, ‘not invented here’, liability, and ownership of components must be solved before the reuse of software will be accepted as a valid business strategy. This paper, however, concentrates on the technical issues relating to software reuse. Since the analogy of software components to mechanical and electronic components has been made, let us look at some of the issues regarding the development and introduction of integrated circuits in the 1960’s for a parallel.

Design handbooks also served as catalogs for integrated circuits show all of the key characteristics necessary for the design engineer. The electrical characteristics included voltage requirements, power dissipation, propagation delays. Table characteristics provide information essential for the reuse of the components. In a similar vein, the software engineer requires certain information about the characteristics of a software component. The characteristics include some indication of the component’s quality including parameters relating to efficiency and trustworthiness. Software differs from hardware, however, in that it will usually not be reused without some adaptation. For this reason characteristics relating to the component’s modularity and/or maintainability ought to be required.

Design handbooks also served as catalogs for integrated circuits. The functionality of the members of each family of components was specified with logic diagrams and truth tables. A parallel exists for software components. McIlroy proposed a simple catalog as a vehicle for defining the components. However, Prieto-Díaz (Ref. 21) showed that a more sophisticated scheme for classification and access ought to be available to support the needs of the software engineer.

The final step for the use of integrated circuits was the synthesis of the components into a usable system. This process was largely left to the engineer, but some computer-aided tools for the automated composition of circuits from functional modules were developed and used. The synthesis of software from discrete components is an area of research where many different approaches have been investigated. Research needs to be conducted into the methodology, techniques, and tools necessary for this synthesis.

SOFTWARE CHARACTERISTICS

Using the hardware component analogy for software reusability, key parameters which characterize a software module ought to be relevant for the inspection and evaluation of software parts. Although the majority of hardware characteristics are not applicable to software, several of them bear close examination. Among these are parameters which measure the quality of software, maintainability, and modularity. This area of measurement is both complicated and controversial. For example, although attempts have been made to model the reliability of software using probabilistic methods, this is one area where the analogy of software to hardware breaks down. Many of the assumptions used to support the concepts of ‘software reliability’ are neither valid nor useful for the purpose of reusability analysis. However, other measures based on physical characteristics and the design process can be applied. Just as with reliability, other characteristics such as maintainability and modularity can best be measured through the use of physical and process characteristics. A hierarchical framework of factors, criteria, and metrics can be defined for the relevant characteristics. This permits an orderly evaluation of the software and influence on its design.

Definition of a Software Evaluation Framework

The use of physical and process characteristics serves as a good model for the establishment of an evaluation framework for the reusability of software. Physical models, measures, and mechanisms used to characterize hardware and its development process do not generally apply to software. The use of probabilistic models and measures based on physical characteristics and the design process can be applied. Just as with reliability, other characteristics such as maintainability and modularity can best be measured through the use of physical and process characteristics. A hierarchical framework of factors, criteria, and metrics can be defined for the relevant characteristics. This permits an orderly evaluation of the software and influence on its design.

Because software engineering is not an exact science, the methods currently available for software design can’t guarantee a trusted product. Even so, analysis techniques can be used to evaluate a design to determine compliance with stated requirements, conformance to design standards, efficiency, and other criteria (Ref. 10). Once a set of objectives is made clear and measurable, software engineers show an amazing ability to achieve those objectives. This is because software engineering is such a complex activity and there are almost an infinite number of choices of how to write any required program. For example, the control structure of a 50 line program with 25 decision statements can be arranged in over 30 million ways (Ref. 13).

The importance of setting explicit and reasonable goals is illustrated by the following experiment (Ref. 17). Five teams of programmers were given the same problem to solve but each had a different objective. The objectives included: minimize the amount of storage, produce maximum output clarity, make the program logic clear, minimize the number of statements, and minimize the amount of time on the project. After the project was complete, a group of experts ranked the resulting programs against each of the objectives. In every case,
the teams achieved their primary objective. No software engineering project, especially one where the software will be continually reused, should be undertaken without a clear set of goals in mind.

Software engineering consists of a sequence of design refinements and each refinement implies a number of design decisions based upon a set of critical factors. At the top level these factors provide a management oriented view of the product. Among the concerns at this level are quality, suitability, and portability. Below these factors are sets of criteria. Examples of these criteria are reliability, complexity, and efficiency. The software engineer needs to consider these criteria in all design decisions and to critically examine and reject potential solutions if they do not meet the system criteria (Ref. 25). Since the design process is iterative, dead ends may be reached. Sometimes the only alternative is to revolve otherwise satisfactory design alternatives and decisions if important criteria cannot be met. The bottom layer of the software engineering framework is a set of software metrics. The software metrics at the bottom level provide the quantitative and qualitative measures which permit these decisions. As discussed earlier, current software metrics are primarily applied to the actual code and are used to measure the sizes of various program objects, the comprehensibility of control structures, and the use of data structure (Ref. 26).

Determination of Evaluation Criteria

Management concerns must be based on three critical factors: suitability, quality, and reusability. A component is suitable if it meets its functional and performance requirements. Suitability can be determined based on the following criteria:

1. Efficiency - A measure of the use of either time or memory space of software
2. Precision - The ability of software to provide answers to the degree of accuracy specified by the requirement
3. Correctness - The ability of software to give the expected results under all combinations of valid input data
4. Software quality is a somewhat nebulous term which refers to the 'totality of features and characteristics of a software product that bear on its ability to satisfy given needs' (Ref. 10). Software quality is used to encompass the following non-functional requirements:
5. Reliability - The probability that software will not cause the failure of a system for a specified time under specified conditions; a function of the inputs to and use of the system (Ref. 10)
6. Maintainability - The ease with which software can be maintained (Ref. 10); affected by complexity
7. Trustworthiness - The ability of software to resist attempts to subvert its purposes by 'unfriendly' users
8. Testability - The ability to partition the software so that an adequate test of its functions can be accomplished
9. Software can be reusable under a variety of circumstances. The reusability factor includes the ability to move code from one type of computer to another, to address slightly different requirements, or to be used in multiple applications. The criteria for the reusability factor are defined below:
10. Generality - The ability of the software to be used without change in a variety of situations; usually with no penalty to pay at run time (Ref. 19) but required for flexibility (Ref. 23)
11. Flexibility - The capability to be easily changed as the software can be used in a variety of situations (Ref. 19)
12. Modularity - A characteristic of systems which are composed of pieces that somehow interact to form a whole (Ref. 19)
13. Understandability - The ease with which a maintainer or reuser can grasp the structure and meaning of software

Software engineering technology has not progressed to the point where machine understanding of a program can even approximate human understanding. In order to characterize and deal with the complexities of software, some alternative representation must be used. Quantitative metrics based on a variety of measurements are convenient to use. These metrics can be estimated easily through examination of the software, they can be used to assess compliance with design requirements, and they can assist in ranking software components according to some factors and criteria for reuse.

Code Metrics

There is a distinction between code and structure metrics. In fact, metrics can be separated into different classes to measure program size, data flow and data structures, and flow of control (Ref. 1). It is important to recognize the differences between these different types of metrics, to learn which factors are measured by the different metrics, and to show how to combine the information from different metrics (Ref. 11). Some of the metrics have been shown to be sensitive to certain changes in software components. For example, the value of some metrics may vary significantly under program transformation or maintenance. Other metrics may not change at all depending on the nature of change. These combinations can indicate problems in certain areas critical to reusability such as modularity. In other cases, values which are 'outliers' don't appear frequently in real-life systems and should lead to close scrutiny of the component even if other metrics appear normal (Ref. 11). The metrics described below have been selected because they are potentially useful for reusability analysis:

Code and Size Metrics

Code and size metrics focus on software modules and require visibility into their internal design. They are
relatively easy to calculate based on the number of lines of code, I/O characteristics, operands and operators, and independence and cohesion of modules (Ref. 22).

1. Lines of code - Program size is easy to calculate by simply counting the lines of code. Unfortunately, there is no generally accepted definition of a line of code. A line of code may include: executable program statements, comments, data declarations, macro instructions, reused code, scaffold code, code changed due to requirements change, test programs, and utility code (Ref. 17).

2. Halstead's Effort metric - This metric is a refinement of measuring program size by counting the lines of code. It is supported by several empirical studies (Ref. 8). The intelligence content of a module correlates well with total programming and debugging time. The intelligence content appears independent of programming language. It increases only as the complexity of the problem not the solution increases. Programming effort, measures the effort required to comprehend an implementation and may be used as a measure of program clarity. It is usually used to estimate the effort to implement a requirement and its understandability (Ref. 20).

3. System features - Although not formalized, two different independence metrics have been used to measure portability effort. The system independence metric counts the number of system references, including utilities, libraries, routines, and other facilities, and the number of I/O references. The machine independence metric is calculated based on the amount of machine dependent code and machine dependent data representations (Ref. 23).

**Data Structures and Data Flow**

These measures assume complexity is influenced by use of data within the program. Although the predicting power of these metrics for software maintenance or reusability has not been formally investigated, they are intuitively pleasing (Ref. 9).

1. Elshoff's Span between data references - This metric is calculated by examining the 'span' between data references within a program. The span is the number of statements between two references to the same identifier. If there are n references to an identifier in a module, calculate the metric as the arithmetic mean of the n-1 span values. Although there are no supporting empirical studies for this metric, it seems logical that errors would be more frequently introduced when use of a variable is logically dispersed throughout a module (Ref. 6).

2. Bassili's segment-global usage pair - Bassili's metric bases program complexity on the use of global data within a module. Compute the metric, relative percentage of actual use (RUP), by dividing the number of times a module actually accesses a global data item by the number of global data items. The larger the value of RUP, the greater the possibility a global value may have its value changed in another segment. This metric correlates with an increased chance of error when the module is modified (Ref. 1).

3. Chapin's Q measure - This measure treats data items differently depending on how they are used (Ref. 4). It is used to compute program complexity by counting the number of data items used to control a segment, needed as input to produce a segment's output, changed or created within a segment, or just 'passed through' without being changed. A value for additional complexity due to data communicated in iterative procedures is computed by adding 2 for every control data item created or modified outside a module (Ref. 2). The Q metric gives an upper and lower bound on complexity but its applicability is uncertain (Ref. 16).

4. Yau and Colloffello's Logical Stability Metric - This metric estimates the stability of a program under change based on ripple effect of variable changes. The measure of stability is inverses of ripple effect measure. Yau claims the stability measure is one of the most important factors affecting software maintainability, modifiability, reusability and ease of redevelop-ment (Ref. 26).

5. Henry and Kafura's Information Flow Metric - This metric is used to measure information flow through a program. It was used to study the UNIX operating system. A correlation coefficient of 0.95 was demonstrated between this metric and errors. Interestingly, it is only weakly correlated to code type metrics (Ref. 11).

**Program Control Structure Metrics**

All of these metrics view a module as a component of a larger system and focus on the interconnection of system components (Ref. 11). Most techniques are based on use of a directed graph to represent the system. The edges of the graph represent a sequential block of code. The edges connect the nodes and represent the flow of control within the program (Ref. 9). These metrics are sensitive to the introduction of loops into programs. They are justified because there are many ways of getting to some of the points in the program and the programmer must account for all of them (Ref. 20).

1. McCabe's Cyclomatic complexity number - McCabe's metric is based on graph theory. The cyclomatic number is calculated from the number of edges, nodes, connected components of a flow graph (Ref. 13). This metric is often used to measure the understandability and testability of a program (Ref. 26). There is also a close correlation between ranking by complexity number and subjective reliability ranking by experts and, more formally, a correlation of 0.96 between metric number and error ranking (Ref. 1).

2. McClure's Control Flow Metric - McClure's metric focuses on complexity associated with control structures and control variables. There is a higher complexity when control variables appear in conditional statements which determine the invocation of other procedures. Data suggests the complexity of each module should be minimized and complexity evenly distributed among modules (Ref. 14).

3. Woodfield's Syntactic Interconnection Measure - This is a hybrid model based on interconnections. It produces very close estimates of actual program development times. Based on a sample of 30 small programs, the metric was able to account for 80 percent of variance in programming time with an average relative error of only 1 percent.

4. Myers' extension to cyclomatic complexity - Myers extended McCabe by recognizing that compound conditions are more complex than simple conditions. The metric gives an upper and lower bound on complexity but its applicability is uncertain (Ref. 16).

5. Gilb's logical complexity - Gilb's metric is a simple measure of the amount of decision making logic in a program. Experimental results show correlation with error proneness, development cost, and development time (Ref. 7).

6. Parnas' uses relation - Although not really a metric, the 'uses' relation is based on graph theory and can provide a useful indication of the proper structuring of modules. The 'uses' relation can be simply defined as a module requiring the presence of a correct version of another module. By restricting the relation 'uses' so that its graph is loop free, circular control problems can be eliminated. It is possible to assign programs to hierarchy levels (Ref. 19). Parnas also suggests there are other criteria which can be used before allowing one program to use another.

7. Chen's measure of program complexity - Chen's metric is another topological complexity measure that is sensitive to nested decision structures. It uses mani-
nal intersect number 'min' of program flowgraph. The
'min' metric is computed as the number of times a line
intersects the edges of the graph when the line is drawn
such that it enters every region of the graph exactly
once. For a given number of decisions, its value is
maximum when every decision is nested. Its value in
minimum when all decisions are serial. Using Chen's
complexity number and assumptions about the structure
of program, a complexity estimate based on the number of
decisions can be computed. The metric is strongly re-
lated to programmer productivity. As it increases,
productivity decreases in a quantized manner (Ref. 5).

MEASUREMENT CRITERIA

The set of metrics defined above are not useful for
the development of reusable software without relating
them to the processes which drive the development pro-
cess. Figure 1 uses a subjective evaluation technique
to relate the quantitative measures defined by the soft-
ware metrics to the reusable software factors and crite-
rria. The metrics are classified into one of three
categories if they show a correlation with any of the
criteria for suitability, quality, or reusability. The
categories are ordered based on the amount of informa-
tion they yield about the design criteria. Each cate-
gory uses a different type of measurement scale (Ref. 9):

1. Nominal Scale - This measure classifies the
items based on groups which are relevant to the design
criteria. For example, a maintainability scale of 'easy
to maintain', 'hard to maintain', or 'not maintainable'
could be used to group systems.

2. Interval Scale - This measure not only ranks
the items to each other but assigns values which tell
how far apart they are. For example, five systems could
be assigned maintainability scales of 10, 27, 55, 80,
and 95. The advantage of this type of scale is that it
does not have to be revised when new software modules
are added.

3. Ratio Scale - This scale not only determines
how far apart the items are, but determines how far they
are from some arbitrary standard. This type of measure
is the most powerful since it gives a direct indication
that one module is twice (or some other value) as main-
tainable as another.

The metrics identified at this point show an
influence over a wide variety of design criteria for
reusability. Some key factors including efficiency,
precision, generality, and expandability are not
measured to any real extent.

DESIGN GUIDELINES

Software engineering is not an exact science. In
fact, it is debatable whether software engineering is a
real engineering discipline. In any case, the state of
the art in software is far behind the state of the art
in other areas of engineering (Ref. 20). There are,
however, some areas of progress. Adaptability of a
program to changes which would make it reusable can be
measured 'primarily in terms of the degree to which it
is neatly structured' (Ref. 25). Some measures sensi-
tive to software structure are available, but the
overall trend does not indicate a simple parametric
solution. Data show software enhancement and repair
activities cause increases in all types of metrics, impact many different aspects of the system simultane-
ously, and cannot keep the effect of changes localized
(Ref. 11).

The software engineer should keep the concept of
designing a family of programs rather than a single
program forefront in his mind. Attention should be paid
the aspects of programs which make them part of a

<table>
<thead>
<tr>
<th>Type of measurement:</th>
</tr>
</thead>
<tbody>
<tr>
<td>'N' - Nominal</td>
</tr>
<tr>
<td>'I' - Interval</td>
</tr>
<tr>
<td>'R' - Ratio</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halstead's Effort Metric</td>
</tr>
<tr>
<td>N</td>
</tr>
<tr>
<td>System Features</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>System Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eishoff's Span</td>
</tr>
<tr>
<td>I N</td>
</tr>
<tr>
<td>Basil's Usage Pair</td>
</tr>
<tr>
<td>I I N</td>
</tr>
<tr>
<td>Chapin's Q Measure</td>
</tr>
<tr>
<td>N I I I I I I I I</td>
</tr>
<tr>
<td>Logical Stability Metric</td>
</tr>
<tr>
<td>B B B B B B B B B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Information Flow Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>McBride's Complexity</td>
</tr>
<tr>
<td>I I I I I I I I I I I I</td>
</tr>
<tr>
<td>McClure's Control Flow</td>
</tr>
<tr>
<td>N I I I I I I I I I I I</td>
</tr>
<tr>
<td>Syntactic Interconnection</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Myers' Extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>I I I I I I I I I</td>
</tr>
<tr>
<td>Gilb's Logical Complexity</td>
</tr>
<tr>
<td>N I I I I I I I I I</td>
</tr>
<tr>
<td>Parnas' Uses Relation</td>
</tr>
<tr>
<td>I I I I I I I I I</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chen's Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>I I I I I I I I I</td>
</tr>
</tbody>
</table>

FIGURE 1 - Reusability Criteria Measurement

206 1988 PROCEEDINGS Annual RELIABILITY AND MAINTAINABILITY Symposium
family. Parnas proposed a few rules for constructing modular systems which are still valuable (Ref. 191):

1. A data structure, its internal linkings, accessing procedures, and modifying procedures are part of a single module.
2. The sequence of instructions necessary to call a given routine and the routine itself are part of the same module (not really applicable to HOLs but to assemblers).
3. The formats of control blocks used in queues in operating systems and similar programs must be hidden within a "control block module".
4. Character codes, alphabetical orderings, and similar data should be hidden in a module.
5. The sequence in which items should be processed should (as far as practical) be hidden within a single module.

The use of a particular design technique by different software engineers does not necessarily lead to consistent modules. In fact, the modularity of systems with a particular technique was more different than alike. Thus, it appears a well-defined methodology is needed before the design of reusable code can be controlled and measured.

SUMMARY OF RESULTS

A framework to evaluate the potential of software to be reused has been presented. The analogy of software to hardware provides a good model to examine reusability. But when looking at specific measures of goodness, a hierarchy for the analysis of software designs leads to a series of measures of the software code. These measures offer potential to control the design for reusability, but no empirical evidence exists at this time. Additional research needs to be done in the areas of:

1. Definition of new metrics which are sensitive to characteristics of the software such as efficiency, generality, and adaptability.
2. Validation of the metrics which do exist and correlation of their influence on important design criteria.
3. Identification of design rules and methodologies which will lead to reusable designs.

REFERENCES

2. L. A. Belady, 'Evolved Software for the 80s'. IEEE Computer, Vol. 12, No. 2, February, 1979, pp. 70-75
15. M. B. McIlroy, 'Mass Produced Software Components', Report on a Conference by the NATO Science Committee, October 7-11, 1968, Garmisch, Germany
16. G. Myers, 'An Extension to the Cyclomatic Measure of Program Complexity', ACM SIGPLAN Notices, October 1977, pp. 61-64
17. G. Parikh, How to Measure Programmer Productivity, Shetl Enterprises, Chibago, 1971

BIOGRAPHY

James A. Hess
US Army Communications Electronics Command
Office of the Technical Director (AMSEL-TOD)
Ft. Monmouth, New Jersey 07703 USA

Jim Hess advises the Technical Director of the US Army Communications Electronics Command on software engineering matters. He lives in Alexandria, Virginia and represents the command on software engineering matters in the Washington, DC area. From 1980 to 1986 he managed many of the Army Material Command’s software engineering activities. From 1976 to 1980 he was the Senior Reliability Engineer at the Army Material Command, and from 1971 to 1976 he performed similar functions at the Army Electronics Command. He is now pursuing a PhD in Information Technology at George Mason University. His Master of Engineering in Industrial Engineering is from Texas A&M University and his BSIE is from Newark College of Engineering.