The measurement of software design quality

James Kenneth Blundell, Mary Lou Hines and Jerrold Stach

Computer Science Telecommunications Program, University of Missouri-Kansas City;
5109 Rockhill Road, Kansas City, MO 64110, USA
E-mail: blundell@cnst.umkc.edu

Software quality involves the conformance of a software product to some predefined set of functional requirements at a specified level of quality. The software is considered valid when it conforms to these "quality factors" at some acceptable level. There are a large number of quality factors against which software may be validated. This paper discusses the development of traditional software metrics in relation to the anticipated structure of a software system. The taxonomy of a software system primarily relies upon the dissection of the software system into modules. Modular design is the cornerstone of quality software, and metrics that can predict an optimum modular structure are critical. By examining the theoretical bases on quality metrics, a base set of common quantitative metrics can be devised and mapped to quality metrics in which they reside. This paper surveys existing metrics and suggests the derivation of software design metrics from software quality factors. Measurable software attributes are identified and suggested as potential design metrics.

1. Introduction

Software quality involves the conformance of a software product to some predefined set of functional requirements at a specified level of quality [Card and Agresti 1987; Cavano and McCall 1978; Mendis 1982]. The software is considered valid when it conforms to these "quality factors" at some acceptable level. There are a large number of quality factors against which software may be validated (table 1). Such lists are common in software development, where the product is evaluated on the basis of its operational characteristics, its ability to undergo change and its adaptability to new operational conditions [Fairley 1985].

Table 1 contains a list of thirty-nine typical software quality measures. Few software engineers would argue against the importance of any software product conforming to, at least, a subset of these factors. However, there would be much less agreement regarding the means by which software should be evaluated against any set of quality metrics.

Considerable research has been directed towards the identification and development of software metrics [Jones 1978]. Metrics have generally been based upon either Halstead's [1977] software science or McCabe's [1976] cyclomatic complexity. However, there are limitations to these metrics which render them impractical and incapable of measuring salient software characteristics. Most impracticality arises from the point
Common software engineering quality measures and their characteristics.

<table>
<thead>
<tr>
<th>Quality Measure</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>• precision of computations {10} and control {11}</td>
</tr>
<tr>
<td></td>
<td>• assessment of freedom from error {8}</td>
</tr>
<tr>
<td>Adaptability</td>
<td>• ease of introduction of new features {5}</td>
</tr>
<tr>
<td>Auditability</td>
<td>• ease of checking conformance to standards {3}</td>
</tr>
<tr>
<td>Availability</td>
<td>• percentage of time that a program is operating according to requirements at</td>
</tr>
<tr>
<td></td>
<td>a given point in time</td>
</tr>
<tr>
<td>Changability</td>
<td>• ease of program change {2}</td>
</tr>
<tr>
<td>Completeness</td>
<td>• degree of implementation of required function {18}</td>
</tr>
<tr>
<td>Conciseness</td>
<td>• lines of code {10}</td>
</tr>
<tr>
<td>Consistency</td>
<td>• uniformity of design and documentation</td>
</tr>
<tr>
<td>Correctness</td>
<td>• program satisfies specification {18} • meets user expectations {18}</td>
</tr>
<tr>
<td></td>
<td>• fault-free {8}</td>
</tr>
<tr>
<td>Data commonality</td>
<td>• use of standard data structures and types {14} {16}</td>
</tr>
<tr>
<td>Dependability</td>
<td>• see Reliability</td>
</tr>
<tr>
<td>Efficiency</td>
<td>• simplify arithmetic and logical expressions {13} {15} • shorten nested</td>
</tr>
<tr>
<td></td>
<td>loops {12} {15}</td>
</tr>
<tr>
<td></td>
<td>• avoid multidimensional arrays {14} • avoid pointers/complex lists {14}</td>
</tr>
<tr>
<td></td>
<td>• use fast arithmetic operations {13} • do not mix data types {16}</td>
</tr>
<tr>
<td></td>
<td>• use integer arithmetic Boolean expressions {14} • minimize I/O requests</td>
</tr>
<tr>
<td></td>
<td>{9} • buffer I/O to reduce communications overheads {4}</td>
</tr>
<tr>
<td>Error tolerance</td>
<td>• damage occurring due to an error</td>
</tr>
<tr>
<td>Expandability</td>
<td>• degree to which program can be extended {2}</td>
</tr>
<tr>
<td>Flexibility</td>
<td>• effort required to modify {2}</td>
</tr>
<tr>
<td>Functionality</td>
<td>• capability, generality and security of system {18}</td>
</tr>
<tr>
<td>Generality</td>
<td>• breadth of application {4} {9}</td>
</tr>
<tr>
<td>Hardware independence</td>
<td>• degree of decoupling from hardware</td>
</tr>
<tr>
<td>Human factors</td>
<td>• quality of the user interface</td>
</tr>
<tr>
<td>Integrity</td>
<td>• control of access to unauthorized users</td>
</tr>
<tr>
<td>Interoperability</td>
<td>• effort required to couple to other programs and systems {4}</td>
</tr>
<tr>
<td>Maintainability</td>
<td>• problem recognition time {7} {17} • administrative delay time {17}</td>
</tr>
<tr>
<td></td>
<td>• maintenance tools collection time {17} • problem analysis time {7} {17}</td>
</tr>
<tr>
<td></td>
<td>• change specification time {2} {17} • active correction time {17}</td>
</tr>
<tr>
<td></td>
<td>• local testing time {6} {17} • global testing time {6} {17}</td>
</tr>
<tr>
<td></td>
<td>• maintenance review time {17} • total recovery time {17}</td>
</tr>
<tr>
<td>Modifiability</td>
<td>• ease of changing a program {2}</td>
</tr>
<tr>
<td>Modularity</td>
<td>• functional independence of program components {9}</td>
</tr>
<tr>
<td>Operability</td>
<td>• ease of operation {7}</td>
</tr>
<tr>
<td>Portability</td>
<td>• effort to transfer the program from one virtual machine to another {17}</td>
</tr>
<tr>
<td>Reliability</td>
<td>• performance of intended function for a given period of time {18}</td>
</tr>
<tr>
<td>Reusability</td>
<td>• ease of reference {7} {9} • standardization {7}</td>
</tr>
<tr>
<td></td>
<td>• ease of integration {4} • reuse of objects {9}</td>
</tr>
<tr>
<td>Robustness</td>
<td>• ability to continue operation despite invalid inputs {18}</td>
</tr>
<tr>
<td>Security</td>
<td>• availability of mechanisms that protect the program/data</td>
</tr>
<tr>
<td>Self-documentation</td>
<td>• degree to which code provides documentation {7}</td>
</tr>
<tr>
<td>Simplicity</td>
<td>• ease of understanding {7}</td>
</tr>
<tr>
<td>Supportability</td>
<td>• ease of extending, adapting and servicing a program {2}</td>
</tr>
</tbody>
</table>

Table 1 (continued)
in the software development process at which the metric may be applied. So-called “code metrics” are essentially irrelevant in the software design portion of the development process, since an implementation must exist before they can be applied. This is clearly too late in the process to properly direct the software design along life-cycle, cost-effective lines.

This paper discusses the development of traditional software metrics in relation to the anticipated structure of a software system. The taxonomy of a software system primarily relies upon the dissection of the software system into modules. Modular design is the cornerstone of quality software, and metrics that can predict an optimum modular structure are critical. By examining the theoretical bases of quality metrics, a base set of common quantitative metrics can be devised and mapped to quality metrics in which they reside. This paper surveys existing metrics and suggests the derivation of software design metrics from software quality factors. Measurable software attributes are identified and suggested as potential design metrics.

2. Traditional software metrics

Most existing metrics are easily computed properties of source code, such as the number of operators and operands, the complexity of a control flow graph, the number of parameters and global variables in routines and the number of levels and interconnections of a control graph. Numeric values are used to express the complexity of the code [Dunsmore 1984; Fenton 1992].

Existing code metrics are not promising. The majority of metrics are presented in such a way that it is unclear what exactly is being measured. In addition, there has been a tendency to create a metric and then search for relevancy and meaning for that metric. Metrics should derive from the need to quantify a software design attribute.

Other problems that exist with metrics are that they cannot be validated, they are not objective, the quality measure is only relative, they depend upon a small set of measurable properties, they do not measure the full set of quality criteria and the metrics may affect more than one quality criterion, resulting in conflicts.

The two most significant achievements in software metrics have been Halstead’s [1975, 1977] theory of software science and McCabe’s [1976] measure of cyclomatic complexity.
Table 2
Halstead’s metrics.

<table>
<thead>
<tr>
<th>Name of metric</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total number of operators</td>
<td>N1</td>
</tr>
<tr>
<td>Total number of operands</td>
<td>N2</td>
</tr>
<tr>
<td>Number of unique operators</td>
<td>n1</td>
</tr>
<tr>
<td>Number of unique operands</td>
<td>n2</td>
</tr>
<tr>
<td>Program length</td>
<td>N1 + N2</td>
</tr>
<tr>
<td>Program length estimator</td>
<td>n1 log₂ n1 + n2 log₂ n2</td>
</tr>
<tr>
<td>Program volume</td>
<td>(N1 + N2) log₂(n1 + n2)</td>
</tr>
<tr>
<td>Language level</td>
<td>(2 * n2)/(n1 + N2)</td>
</tr>
<tr>
<td>Program effort</td>
<td>V/L</td>
</tr>
</tbody>
</table>

Halstead hypothesized that algorithms may be characterized by a set of invariant laws, much like physical science [Fitzsimmons and Love 1978]. This is noteworthy since the original approach was for analyzing algorithms not programs [Shen et al. 1983]. Algorithms are language independent but the number of lines of code that implement an algorithm is language dependent. Software science is based upon measures that can be computed at compile time [Fitzsimmons and Love 1978]. A serious disadvantage of using the number of lines of code as a metric is its unavailability at design time. Metrics cannot just count lexical tokens but should consider the structure created [Henry and Kafura 1981].

Counting the number of lines of code is intended to be an elementary metric, however, the implementation language reduces the precision of the metric. Code metrics tend to be oriented towards a specific language or application. A further complication is the ease by which a line of code can be defined. Most high level languages have “code overhead” that must be used in order to implement a basic algorithm. Difficulties therefore exist in defining what constitutes a “line of code”. In fact, there is little standardization in obtaining an accurate value for any of the basic inputs for Halstead’s equations.

Table 2 summarizes the original metrics as defined by Halstead. Numerous authors have validated [Albrecht and Gaffney 1983; Bailey and Dingee 1981; Basili 1990; Fitzsimmons and Love 1978; Gordon and Halstead 1976; Kafura and Canning 1985], applied [Christensen et al. 1981; Cook 1982; Curtis et al. 1979a; Gilb 1985; Grady 1994; Grady and Caswell 1987; Henry and Kafura 1984; Ince 1991; Kitchenham et al. 1990; Li and Chung 1987; Lind and Vairavan 1989; Myrvold 1990; Samson et al. 1987; Schneidewind and Hoffman 1979; Walston and Felix 1977; Woodfield et al. 1981] or critically examined [Card and Agresti 1987; Hamer and Frewin 1982; Ince 1990; Lassez et al. 1981; McCarthy 1962; Perlis et al. 1981; Shen et al. 1983; Shepherd 1990] these software metrics or the elements of software science. Although some metrics have viable application, such as the program effort metric in relation to software maintenance, there is an underlying uneasiness relating to the theoretical underpinnings of Halstead’s software science [Shen et al. 1983]. Metrics such as coding time and coding errors are not independent. A software engineer can code
faster but make more mistakes. Development times in software metrics assume coding is 100% of effort and 100% concentration [Hamer and Frewin 1982], that is, there is a considerable overhead on a programmer’s time.

Traditional software metrics also predict an increase in complexity measures when code length is increased. In fact, increasing the number of lines of code can reduce the number of connections between system components, thus reducing complexity. This additional code reduces information flow complexity. Therefore, a larger but better structured system may be less complex than a smaller poorly structured one [Henry and Kafura 1984].

McCabe’s cyclomatic metric [McCabe 1976; Waguespack and Badlani 1987; Weyuker 1988; Zuse 1991] correlates the difficulty of understanding a program to a numerical evaluation of the complexity of the control flow graph [Baker and Zweben 1980; Chapin 1979; Chen 1978; Curtis et al. 1979a; Kearney et al. 1986; McClure 1978; Munson and Khoshgoftaar 1990; Navlakha 1987; Zolnowski and Simmons 1981]. A flow graph is a directed graph containing a set of nodes and edges. Control complexity is related to the number of edges in the flow graph.

The cyclomatic number, \( V \), of a connected graph, \( G \), is

\[
V(G) = E - n + 2p,
\]

where \( E \) = number of edges, \( n \) = number of nodes, \( p \) = number of connected components.

McCabe’s metric of cyclomatic complexity counts the number of basic control path segments [Curtis et al. 1979b; Sunohara et al. 1981], such a number being easy to compute [Elshoff and Marcotty 1978; Hansen 1978]. The cyclomatic number can be determined prior to any code being written and identifies the number of independent substructures [Schneidewind and Hoffman 1979]. Cyclomatic complexity recognizes that compound predicates increase program complexity [McCabe and Butler 1989; Myers 1977] and there is evidence to suggest a connection between decision nodes and complexity [Allen and Cooke 1976; Bail and Zelkowitz 1978; Davis and LeBlanc 1988; Gill and Kemmerer 1991; Kafura and Reddy 1987; Oviedo 1980; Woodfield 1979; Woodward et al. 1979]. This metric has been applied to software maintenance [Kafura and Reddy 1987] and testing [Banker et al. 1989; Tai 1980] since it can detect all feasible paths through an algorithm and can evaluate the reachability of nodes [Andersson et al. 1994].

Neither McCabe nor Halstead consider nesting as additional complexity [Curtis et al. 1979b; Shepperd 1988]. For example, three loops in succession can result in metric values identical to those for three nested loops.

In essence, Halstead’s metrics focus on measurable attributes within functions and modules, while McCabe’s can be used to measure information flow both within a module and between modules. In relation to design metrics, McCabe’s is conceptually more relevant to design and traditional McCabe metrics have been tested as potential design metrics [McCabe and Butler 1989].
3. **Program taxonomy**

The Bohm–Jacopini theorem of "program design" states that: "any single entry, single exit program segment that has all statements on some path from the entry to the exit can be specified using only sequencing, selection and iteration" [Bohm and Jacopini 1966].

Programs can be decomposed into a common structure of input, process and output. The number of jumps, loops and selections increases the difficulty of understanding the program and elevates the software complexity [Gordon 1979]. High-level programming languages are designed to make the implementation of algorithms easier to understand and less complex. Information flow is a measure of program complexity that exists before implementation [Kafura and Henry 1981]. Two modules or program units are connected, if there is a flow of information between the two.

A given high-level language defines a process by the use of a set of operators and operands. The vocabulary of a program is considered to be the sum of the number of unique operators and unique operands the program contains [Grier 1981]. Such operators and operands are collectively referred to as "tokens" [Shen et al. 1983]. Tokens may be considered to be atomic programming units.

Programs are hierarchical and may be viewed at a variety of distinct levels of aggregation or specialization. A line of code is a collection of tokens. A program unit is a collection of lines of code. A module is a collection of program units and a program is a collection of modules.

Early investigations focused on the internal organization of individual programs rather than software system structure. Generally, a large unit of code or a single unit program should be discouraged and the program should be implemented as a collection of integrated modules. The decomposition of a program into modules produces an interconnected, hierarchical structure. This interconnection of modules adds to the system complexity as does the complexity of individual components [Card and Agresti 1987]. System complexity therefore derives from both inter-modal and intra-modal complexities.

For a given design, a large, hierarchical design structure indicates a high degree of modularity and a high level of inter-modal complexity. A lesser number of modules produce lower inter-modal complexity but higher intra-modal complexity. In architectural design, complexity in a high level module can be deferred to a module that is lower in the hierarchy. Modules that are situated lower in the architectural hierarchy will tend to have lower intra-modal complexity, at the expense of higher inter-modal complexity.

4. **Modular design**

Modular design involves distributing the functional requirements of a software project across a number of modules. Modularity is the quality that measures the relative success of the decomposition of the software system into modules and sub-functions.
Structural complexity then derives from the relationships between these resultant system modules [Parnas 1972]. The simplest structural complexity derives from a single module, however, internal complexity of a single module will be maximized. When a decomposition produces modules of equal functionality, then the designed modules should have equal internal complexity.

Two principal qualitative measures of module design are coupling and cohesion. Coupling measures the simplicity of the connection between modules while cohesion measures the singularity of function of a single module. To minimize coupling, system modules should have small interfaces, which are derived from a minimum set of passed parameters or shared data [Lohse and Zweben 1984].

Coupling is an influential design principle of software quality. Reducing the number of modular interconnections improves software quality. The work performed by a module is related to the number of inputs and outputs (its level of coupling) which affects its internal complexity (its level of cohesion).

Cohesion or average modular strength is a measure of local or internal modular complexity. Such module strength derives from the strength of the connections between program units. The design of modules, in relationship to cohesion, benefits from what is termed cluster analysis. In this process, modules are grouped by similar couplings and mutually shared data. A matrix of data bindings between procedures produces a dissimilarity matrix. Clustering then groups those procedures that have the smallest dissimilarities, as potential candidates for logical modules. Cohesion is an accepted measure of intra-modular quality while coupling is a pertinent measure of inter-modular complexity.

Two other metrics address the inter-modular complexity of a system design. These are termed fan-in and fan-out. The fan-out from a module is the number of calls from that module. The square of fan-out is a complexity measure and a design metric that correlates to the probability of defects. A module with zero fan-out adds nothing to the structural complexity. Large fan-out indicates that there may be a missing design layer, that is, the hierarchy of modules is too shallow.

Fan-in describes where a single module is called by collection of modules that are immediately above it in the design hierarchy. Fan-in is not an important complexity discriminator.

In modular design, cohesion should be maximized while coupling is minimized. This implies that over-modularization may be as bad as under-modularization. Design errors increase with modules having high coupling, low cohesion, high complexity and large size. Also, as the number of modules increases so does the need for more statements that provide linkage protocols.

5. The measurement of software quality

The process of developing quality software can be simplified by viewing the process as either ensuring software is right the first time or that it gains conformance by the removal of quality deviations through the development process. Error free code,
naturally, takes longer to originate but equal quality could be attained by rapid design and coding followed by removal of errors through rigorous testing and correction [Kafura and Canning 1985].

Errors are generally attributable to either design, where the program fails to properly implement the specification, or programming, where errors exist due to coding mistakes. Research has been inconclusive in finding the principal source of error [Abe et al. 1979; Basili and Perricone 1984; Lipow 1982]. Most non-clerical errors can be attributed to design problems, where the subsequent implementation is incomplete, erroneous or deviant, with respect to the requirements specification [Schneidewind and Hoffman 1979]. The majority of these errors could have been detected during design [Troy and Sweden 1981].

If a programmer is careful in the design phase a program will be easy to understand, will contain few errors, and should be easy to correct, if an error is discovered [Jayaparakash et al. 1987; Shen et al. 1983].

Errors tend to originate due to the complexity of the software or the amount of effort needed to code a large project. Higher level languages have been developed to reduce this programming effort. Characteristics of program vocabulary have been used to discriminate between reliable and error-prone software [Bowen 1979]. Analysis has shown that the number of errors per line of code increases with the number of lines of code [Lipow 1982], the number of operators and operands in a program [Fitzsimmons and Love 1978], and the number of code segments. A code segment is defined as a sequence of executable statements that must all be executed [Lipow 1982]. The previous section presented only two quality measures, coupling and cohesion.

However, there are clearly structural characteristics of programs that reflect the difficulty of producing error free code and the difficulty of detecting errors during debugging and testing. Such characteristics may be used as metrics for program design purposes.

We will proceed to present those metrics and their characteristics (table 1) and then to associate these quality metrics with known software characteristics via table 3. An entry of ‘1’ in table 3 indicates a correspondence between a software characteristic and a design attribute. Column 2 of table 3 maps those correspondences to characteristics of quality measures in table 1. Table 3 indicates a number of design attribute sets covering the entire set of software characteristics. Since these attribute characteristics relate directly to the quality characteristics of table 1, we surmise that some cogent set of quality metrics might extend from design attributes via formal or informal translations.

The set of software characteristics in table 3 is entirely covered by the design attribute pair (COM,ITA). Suppose the cover set is feasible. (COM,ITA) will map to a set of software engineering quality measures in table 1. The overlap of their vectors suggests translation pairs: (COM,ITA)/Ease of testing, (COM,ITA)/Ease of understanding, (COM,ITA)/Freedom from error, and (COM,ITA)/Ease of maintenance.

In this paper, we are suggesting that the density of ‘1’ entries in table 3 and the correspondence of those entries to the quality characteristics in table 1, suggest
Table 3
Software characteristics and their design attributes. Design attributes: COH = Cohesion, COM = Program complexity, COU = Coupling, DAS = Data structures, ITA = Intra modular complexity, ITE = Inter modular complexity, TOS = Token selection.

<table>
<thead>
<tr>
<th>Software characteristics</th>
<th>#</th>
<th>COH</th>
<th>COM</th>
<th>COU</th>
<th>DAS</th>
<th>ITA</th>
<th>ITE</th>
<th>TOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conciseness</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ease of change</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ease of checking conformance</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Ease of coupling to other systems</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Ease of introduction of new features</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Ease of testing</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Ease of understanding</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Freedom from error</td>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Functional independence of modules</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Precise computations</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Precise control</td>
<td>11</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Shortest loops</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Simplest arithmetic operators</td>
<td>13</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Simplest data types</td>
<td>14</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Simplest logic</td>
<td>15</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Standard data types</td>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Ease of maintenance</td>
<td>17</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Functional specification compliance</td>
<td>18</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

research should proceed relative to the feasibility of a traceable set of qualitative metrics extending from quantifiable design metrics. The design metrics we feel are candidates are presented in the following section.

6. The metrics of design

Design is the earliest stage in which the system structure is clearly defined. Design generally refers to architectural design, that is, the process of partitioning the required functionality and data of a software system into parts that work together to achieve the full mission of the system. Design first concentrates on perfecting the software architecture and then focuses on the details within modules. Structured design methodology has had a great influence on product quality [Yourdon and Constantine 1978]. Techniques such as information flow analysis and software entropy functions [Mohanty 1979] can be used at design time. Such entropy functions are based upon communication-information theory which decomposes a program structure into distinct classes of sub-systems. Existing entropy metrics can then measure the information content of a given structure. Observing the information flow can lead to measures of complexity, coupling, hierarchical modular interactions and system stress points.

A design tree is an ordered hierarchy of system modules based upon functional decomposition. At each level of the decomposition, the designer must either implement functionality at that level or defer some to a lower level by invoking new modules.
Deferring functionality decreases local complexity but increases architectural complexity. Such a design tree would enter at the top, execute lower-level modules and then exit at the top.

Assessment of software design quality is a high priority [Beane et al. 1984; Szulewski et al. 1981; Yin and Winchester 1978] and design and specification metrics are needed to provide early feedback on the success of the design process. Design metrics can be used to rework a design to avoid anticipated problems [Kitchenham 1990].

Design metrics may be considered to have one or more of three separate functions:

1. They may be used to evaluate the current levels of various software qualities within the project.
2. They may be used to partition the design into an acceptable set of modules.
3. They may be specifically developed to work at given points in the design process.

Measuring software quality is a principal task of software metrics. The following technique has been developed and is shown here as an example of a feasible methodology for the establishment of valid metrics for software qualities.

In table 1, software qualities are listed together with typical quality factors that describe in more detail, the definition of that software quality. A number of the qualities in table 1 measure the quality of the software development process and do not relate to the quality of the software product which directly influences the quality of the design.

Each quality factor that is potentially relevant to product quality, is followed by one or more numbers in braces. These numbers are indices of software characteristics that are shown in table 3. Each quality factor in table 1 has been shown to be comprised of one or more software characteristics. Table 3 lists these characteristics and the seven design attributes that positively contribute to their existence.

Briefly, here is how each attribute may be measured.

- **Cohesion (COH).** Cohesion measures the singularity of purpose of a module. Suitable metrics would relate to the uniqueness of the input data types to the module and the semantic similarities between the function contained within the module.

- **Coupling (COU).** Coupling is maximized when the inter-connectivity of a module is minimized. McCabe complexity metrics are currently most relevant.

- **Data structures (DAS).** The choice of data structure significantly affects the quality of a design. Specification languages may be used to express optimum data types based upon functional requirements. Optimal data structures should be derived from intelligent systems based upon experience analysis.

- **Intra- (ITA) and inter- (ITE) modular complexities.** McCabe complexity measures are still the most pertinent solution given a particular module design.

- **Program complexity (COM).** A measure of problem complexity needs to be derived from a functional specification. Currently, there is no method of measurement that establishes the "degree of difficulty" of a problem. Software entropy is feasible as
a measure of the information content of a problem description and if alternative designs can be similarly rated, then the design which causes the minimum increase in entropy could be measured.

- **Token selection (TOS).** Token selection relates to the number and variety of data values and functions needed for a given solution. Such a metric would be a measure of problem size and its translation into a given design solution.

Each characteristic is listed in table 3 and the attributes that are relevant to that characteristic have been identified. Each attribute is a potential metric, given the above descriptions that the attribute may be evaluated during the evolution of a software design.

This is the basis for an audit trail. Attributes are measurable quantities of software characteristics [Hines and Goerner 1995]. Software qualities have quality factors that in turn are defined by a set of software characteristics [Hines and Goerner 1995]. However, some software characteristics appear in more than one software quality. For example in table 1, *Ease of Change* \( \{2\} \) is a characteristic of changeability, expandability, maintainability, modifiability and supportability. Each software quality measure has a collection of software characteristics, each of which has a set of design attributes, shown in table 3. Each software quality measure may be considered a collection of design attributes which are transmitted through software characteristics.

Table 4 produces an attribute count for each product-related software quality. For example, the attribute count for the quality “accuracy” is a combination of the attributes derived from the characteristics of “accuracy”. These are “freedom from error”, “precise computations” and “precise control”. Such attribute matrices can be developed to show the relative weightings of the attributes and there would seem to be fruitful research in the refinement of characteristics and their attributes.

A second area for design metrics is in the structural partitioning of the system. Structural complexity is a function of the work performed within modules in addition to the connections between modules. This can be minimized by eliminating structural and local components [Card and Glass 1990]. Minimizing structural complexity means minimizing the fan-out from each module and evenly distributing the structural complexity across all modules. On the other hand, local complexity can be minimized by maximizing fan-out. System complexity derives from the complexity within each part and the complexity between each part and, if properly measured, can be used to control the partitioning of systems into more manageable pieces.

There is a trade-off between structural, or intra-modular, complexity and local, or inter-modular, complexity. However, metrics should ensure that the overall system complexity is minimized, since

\[
C = S + L,
\]

where \( C \) is the overall system complexity, \( S \) is the structural complexity, and \( L \) is the local complexity. Such an equation is dependent upon the number of modules in a given system.
Table 4

<table>
<thead>
<tr>
<th>Software quality</th>
<th>Characteristics</th>
<th>Inter</th>
<th>Intra</th>
<th>Cohesion</th>
<th>Complexity</th>
<th>Coupling</th>
<th>Data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td>{8} {10} {11}</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Adaptability</td>
<td>{5}</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Auditability</td>
<td>{3}</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Changeability</td>
<td>{2}</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Completeness</td>
<td>{18}</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Conciseness</td>
<td>{1}</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Correctness</td>
<td>{8} {18}</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Data commonality</td>
<td>{14} {16}</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Efficiency</td>
<td>{4} {9} {12} {13}</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>6</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 1 shows a notional graph that displays the reduction in intra-modular complexity as the number of modules increases. Naturally, the inter-modular complexity grows as the number of modules increases. This implies there will exist an optimum number of modules for a given software system. It is not critical that the exact minimum be identifiable. Rather, it is more important to establish the range of module count, within which the optimum will lie. It is also apparent that the total amount of code in the system will increase as the number of modules increases. This is due to the “code overhead” used to maintain communication between modules and provide error checking for information flows. Such “code overhead” can add 30–50% more lines of code. The optimum number of modules may also be affected by metrics that are specific for desirable properties of the software.

The final type of metric could be those designed for specific measurements at particular points in the design process.
The design quality of a system is too multi-faceted to be measured by one metric. Also, different metrics may need to be applied at different points in the design lifecycle. The metrics of design must measure more abstract features such as function and structure.

Software metrics that provide feedback to programmers during development should be available early enough in the life cycle to guide software testing and maintenance. They should be able to identify modules with the greatest error potential, presumably based upon complexity measures. They should help partition a design to minimize couplings between modules, be able to provide design hints and facilitate a logical choice of algorithms [Baker and Zweben 1979]. They should be capable of optimizing the program architecture while the program is being designed.

Metrics may be designed to evaluate specific functional requirements such as execution characteristics, goal characteristics and syntax characteristics. Execution metrics may measure run-time characteristics such as performance and fault tolerance. They could also estimate “stress points” in a system where there is a potential for high traffic flow. Goal metrics could measure how well the software achieves its functional specification. Metrics could be used to predict the more difficult modules in terms of complexity, identify modules that lack functionality and track the retention of functionality during the design and development process. Goal metrics should account for the computing environment, the application area, the algorithms implemented, the desired levels of reliability and efficiency, and the characteristics of product users. Goal metrics can also detect possible degradation of the software by subsequent maintenance activities.

Syntax metrics would measure the source code and its implementation. These are akin to code metrics that could detect inadequate refinement of modules. This would
be evidenced by too many lines of code or a large fan-in or fan-out at some level of
the module hierarchy. Syntax metrics should also be able to predict the ultimate size
of the software from a requirement’s specification [Basili 1990].

7. Object-oriented design metrics

It would be remiss to discuss design metrics without considering object-oriented
design. Abstraction and encapsulation are the principal design criteria. Abstraction
involves the recognition of similarities between objects and processes in the real world.
Encapsulation bundles data and methods together.

Object-oriented design can produce improved system architectures. For example,
information hiding should reduce variable repetition and local complexity.

Code metrics are not useful metrics in object-oriented programming and cyclomatic complexity is of less interest due to polymorphism.

Object oriented design metrics traditionally include:
• the number of classes and their reuse;
• the degree of multiple inheritance;
• the number of operations which respond to a message;
• the width and height of the class hierarchy;
• the number of operations per class;
• the number of objects and their reusability.

The fact that object oriented design is ontologically based is in conflict with the
notion that modularization emerges from some formal decomposition of the system
design. Booch [1986] defined the notion of object-oriented design around objects, their
attributes, methods and communication patterns. Object orientation cannot separate
data and operations, so complexity measures based upon control structures and data
flow [Gordon 1979] are not relevant. Furthermore, the encapsulation barrier of ob-
jects thwarts the annotation of complexity by measures suggested in [Dunsmore 1984;
Fenton 1992]. For objects, the cyclomatic number and notion of independent control
structures [Schneidewind and Hoffman 1979] have meaning only within the object, and
are not revealed by the public interfaces which invoke internal methods. Halstead’s
metrics focus on measurable attributes within modules and functions. Since objects
are fully encapsulated, function cannot be deferred to some lower level abstraction.
Invariant laws hold within objects but do not pervade an object system.

Object-oriented design is fundamentally a class based exercise. Class design is at
a higher level abstraction than the data and procedural levels considered by metrics we
discussed in earlier sections of this paper. The limitations of those metrics were clearly
recognized with respect to software quality. Pfleeger and Palmer [1990] developed a
cost model around the enumeration of objects and methods. Moreau and Dominick
[1989] offered a few metrics for graphical objects. An initial cohesive approach was
articulated by Lieberherr et al. [1988] as the “law of Demeter”. However, a formal
basis for object based metrics required a grounding of common measurements in some object theory. To that end, Wand and Weber [Wand 1987; Wand and Weber 1990] were able to formalize the system notion of objects by extending Bunge’s ontology [1979]. Specifically, as in [Chidamber and Kemerer 1991]:

- **object coupling** is defined as the effect of one object on the chronologically ordered states of another object over time,
- **object cohesion** is defined to be the intersection of the sets of instance variables of the methods of an object,
- **object complexity** is defined as the degree of cardinality of the properties of an object and
- **object class** is defined as a set of objects sharing a set of common properties. This facilitates the notions of inheritance hierarchy and depth of inheritance.

From these definitions, Chidamber and Kemerer [1991] proposed a theoretically based metrics suite for object-oriented design:

- **Weighted methods per class** (WMC): given a class C with Methods $M_1, \ldots, M_n$ of static complexities $c_1, \ldots, c_n$, $WMC = \sum_{i=1}^{n} c_i$.
- **Depth of inheritance tree** (DIT): the number of ancestor classes which can potentially affect this class.
- **Number of children** (NOC): the number of immediate sub-classes subordinate to a class in the class hierarchy.
- **Coupling between objects** (CBO): the number of non-inheritance related couples with other classes. Object coupling is not associative.
- **Response for a class** (RFC): $RFC = |RS|$ where RS is the response set for a class and $RS = \{M_i\} \supseteq \bigcup_i \{R_i\}$ where $M$ is the set of all methods in the class and $\{R_i\}$ is the set of methods called by $M_i$.
- **Lack of cohesion in methods** (LCOM): let C be a class with methods $M_1, \ldots, M_n$ as before. Let $\{I_i\}$ be the set of instance variables used by method $M_i$. There are $n$ such sets $\{I_1\}, \ldots, \{I_n\}$. $LOCM = \text{the number of disjoint sets formed by the intersection of the n sets}$.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Object definition</th>
<th>Object attributes</th>
<th>Object communication</th>
</tr>
</thead>
<tbody>
<tr>
<td>WMC</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DIT</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOC</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RFC</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CBO</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>LCOM</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
Chidamber and Kemerer [1991] map these to the elements of object oriented design as shown in table 5.

8. Conclusions

Software quality is driven by a set of software characteristics with a potential set of measurable attributes. In this paper, a feasible set of attributes has been identified which may be measured at design time. These metrics may then follow the design through its various stages of development and can track the retention of quality by reference to the current values of the metrics. The metrics may be customized for a particular project in which the software qualities could be weighted for a given design domain. The goal of the proposed research is to establish a traceable set of quality metrics which extend from design attributes by formal or informal translations. It is feasible that a control panel for design could track critical metrics as the software engineer drives the design to completion. Quality software needs a library of reliable, reusable modules with known fault tolerances [Boehm et al. 1976] and only with a viable set of design metrics can such a goal be attained.

Three theoretical bases of quality metrics have been identified from the literature and it appears that these yield a base set of common quantitative metrics. In this paper, these metrics have been mapped to the quality metrics in which they are embodied.

Furthermore, the quantitative metrics are expansive and non-orthogonal to both life cycle phases (i.e., the different entities produced in different phases) and to the quality metrics. This distilled set of theoretically based metrics extends to different software design and development techniques – modular design by system function, modularization by decomposition of computational function or object orientation.

The implication is that there is some set of common quality metrics that can be applied to the life cycle which can be managed quantitatively in each phase – i.e., the solution space for the matrix is defined by the acceptable targets of the previous phase and can be further limited but not expanded in some subsequent phase. This means that in the case of procedural programming and design by decomposition, function may in some cases NOT be deferred. The implication for object orientation is that the grain size of objects may have to be managed relative to measures such as the depth of the inheritance tree, the coupling between objects and the lack of cohesion in methods.

Further work needs to be done to:

(a) define this set of common, theoretically based metrics in terms of the entities produced by each life cycle activity,

(b) quantify the measurement for each entity and life cycle activity and

(c) establish a “traceability principle” for these quality metrics across the life cycle.

Finally, an operations research problem is implied in the need to compute the solution space of the quality metrics for a system and optimize that space across the
outcome of each life cycle activity. If this is attainable then the optimum system for a given application could be established beyond doubt.

References


Chapin, N. (1979), A Measure of Software Complexity, AFIPS Press, College Park, MD.


