Transactions on Information and Communications Technologies vol 8, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

Metrics applicable to software design

X. Yu & D.A. Lamb
Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada

Abstract

Applying metrics to software is a way to measuring and improving software quality. Many metrics apply to software implementations (code), so they cannot be used early in the life cycle. We survey eight modularity and structural complexity metrics applicable to software designs, and summarize the results of empirical validation studies when they are available. We present a data model from which all these metrics can be computed. Aspects of each metric require further study and refinement. However, using some of them during design may still be beneficial.

Introduction

The 4Thought project [14], which aims to develop generic tools to assist software designers, views a design as a database of information about software components and relationships among them. As part of this effort, we have looked for software design metrics that assign numbers to designs or parts of designs, attempting to characterize product attributes (such as maintainability, defect rate, or development time) at an early stage of the life cycle. When we started, we hoped to find a collection of metrics computable purely from design-level information (avoiding code-level properties), applicable to designs intended for implementation in modern modular imperative programming languages, such as Ada, Modula, Turing, or C++. This paper reports on the metrics we have found so far, and presents a first cut at a data model for the information they require.

Few metrics fit exactly what we wanted, so we broadened our search to include some meant for Structured Design (whose concepts do not exactly match our languages), and some measures that depend on code-level properties (such as code length, or detecting which parameters are used to decide flow of control, or whether the code embodies assumptions about the values of a parameter). Such measures, though problematic from our perspective, might be useful in analysing

Author(s)	Metric	Date
Mixed inter- and intra-modular metrics		
McClure	program complexity	1978
Chapin	Q	1979
Henry and Kafura	information flow	1981
Yau and Collofello	design stability	1985
Card and Agresti	system complexity	1988
Shepperd	IF4	1990
Purely inter-modular metrics		
Yin and Winchester	network complexity	1978
Benyon-Tinker	software complexity	1979

Table 1: Metrics Surveyed

legacy code, and might inspire other researchers to develop new metrics that avoid code-level properties.

Work on design measures can be traced as far back as Alexander's book [1]. Current design metrics primarily concern structural complexity. Their development proceeded via Parnas' data abstraction and information hiding [13] and the work of Stevens et al. at characterizing system design complexity [19].

Validation of a design metric is the process of ensuring that the measure reasonably characterizes some accepted design quality attributes. Correlation analysis can be used for investigating statistical relationships between design metrics, in which design change is accompanied by change in the measured result. Correlation coefficients present the strength of a relationship between a design metric and an accepted measure, such as defect count.

We first overviews some design metrics, their interpretations and, where we could find them, results of validation efforts. After presenting our data model, we discusss some aspects of the metrics that need further investigation.

Overview of Design Metrics

Shepperd [15] classifies design metrics into families of purely intra-modular, mixed inter- and intra-modular, and purely inter-modular ones. Purely intra-modular metrics are not true design metrics, as the data needed to calculate them are not available until coding is complete. Table 1 summarizes the surveyed metrics. The following surveys each of the design metrics in turn. Section 2.8 derives a data model summarizing the information needed to compute these metrics.

Much of this literature uses the word "module" to mean what modern programming languages would call a procedure. We have reserved the word "module" to mean some collection of procedures and variables.

McClure's Program Complexity Metric

McClure's metric [12] concentrates on the complexity of control structures (iteration and selection) and control variables used to determine invocation of procedures. A control variable is one whose value directs the path of a procedure invocation. The complexity of procedure i is:

$$P(i) = a_i \times C_a(i) + d_i \times C_d(i),$$

where

a_i – number of procedures directly invoking procedure i,

$C_a(i)$ – complexity of control structures and control variables used in the procedures that invoke procedure i,

d_i – number of procedures directly invoked by procedure i,

$C_d(i)$ – complexity of control structures and control variables used in procedure i to direct invocation of procedures.

To compute these functions we need to know those control variables referenced or modified, and those used in selection or iteration control structures. Detail of how to compute these functions are not included here due to space limitation. McClure's criterion is that the complexity of each procedure should be minimized; meanwhile, the complexity among procedures should be evenly distributed.

Chapin's Q Metric

Chapin's measure of software complexity [6] concentrates on the roles of input and output data of modules in a Structured Design; we have taken the liberty of reinterpreting his work in terms of procedures, parameters, and global variables. To improve the validity of his metric, Chapin recognizes four roles for data:

- "P" data: input data needed for processing;
- "M" data: data changed, created, or modified;
- "C" data: data used to choose functions to perform;
- "T" data: data may pass through a function unchanged.

C data contributes the most complexity. M data is major contributor. P data contributes some complexity. T data passing through a function contributes the least to software complexity.

Chapin's algorithm for computing his metric reduces to the following formulas for the Q_i, the complexity of procedure i, and Q, the program complexity:

$$Q = \sum_i Q_i / n$$
$$Q_i = \sqrt{W_i \times R_i}$$

where

n is the number of procedures in the program,
$W_i = 3 \times C_i + 2 \times M_i + P_i + (T_i/2)$
C_i, P_i, T_i, M_i are the number of input data items to procedure i in sets C and P, the number of input or output data items in set T, and the number of output data items in set M
$R_i = 1 + (E_i/3)^2$

Computing E_i requires determining which procedures contain exit tests for iterations where subordinate procedures are part of the iteratively-invoked loop body. For each such procedure, examine the C data items to determine those used in the exit test. Determine where these C data come from. Add 0 to E for each such C data item if these C data are constants, or come from within this procedure only. Add 1 to E for each such C data item if these C data come from the subordinate procedures that are part of the loop body. Add 2 to E for each such C data item if these C data come from outside the loop body. The reason for including iteration control and distinguishing C data items is that iteration control is believed to be psychologically most difficult aspect of software complexity.

A relatively even distribution of the complexity is also desirable among the procedures of a system. Chapin suggests that Q rarely exceed 11 for procedures with fanout and rarely exceed a Q of 5 for procedures without fanout.

Henry and Kafura's Information Flow Metric

Henry and Kafura's three information flow metrics [7, 11] consider that the prime factor determining structural complexity is the connectivity of a procedure to its environment. They have validated their metric empirically by correlating it with maintenance change data for UNIX.

Henry and Kafura's metrics require the following definitions:

- A global flow exists from procedure A to procedure B if A deposits information into a global variable and B retrieves information from the variable.
- A direct local flow exists from procedure A to procedure B if A invokes B.
- An indirect local flow exists either (assuming procedure A calls procedure B) from B to A if B returns a value to A and A subsequently use it (case 1), or (assuming procedure A calls procedure B and procedure C) from B to C if B returns a value that is passed to C (case 2).

The complexity of a procedure takes into account the complexity of its code (its length in lines), and the complexity of its connections to its environment (its *fan-in* and *fan-out*). Fan-in is the number of local flows that end at the procedure, plus the number of global variables from which the procedure retrieves information. Fan-out is the number of local flows that emanate from the procedure, plus the number of variables it updates. The complexity for a procedure p_i is:

$$p_i = \text{length}_i \times (\text{fan-in}_i \times \text{fan-out}_i)^2,$$

where length_i is the number of line of source code of procedure i, and fan-in_i and fan-out_i are fan-in and fan-out (as defined above) of procedure i.

The multiplication of fan-in and fan-out represents the total possible number of combinations of an input source to an output destination. The squaring of the fan-in and fan-out is based on the belief that the complexity is more than linear in the connections that a procedure has to its environment.

Henry and Kafura use the procedure complexities to establish module complexities. In their terminology, *module* means the set of all procedures that reference a particular global variable. They define the a module's complexity as the sum of complexities of the procedures within it, namely, $\sum_{i=1}^{n} p_i$

Another metric for a given module is the number of global flows possible among the procedures through the global variable. Its formula is:

$$W \times R + W \times RW + RW \times R + RW \times (RW - 1)$$

where R, W and RW are the number of procedures in the module that only real from, only write to, or both read from and write to the global variable, respectively.

We can use the procedure, module complexities and the number of global flows to locate possible design and implementation problems.

Procedure complexity (high fan-in and fan-out) can show three potential problem areas:

- The procedure may perform more than one function.
- The procedure may be difficult to change because of the many potential effects on its environment.
- A missing level of abstraction in the design process; we may need to divide the procedure into two or more separate ones.

The module complexity and the number of global flows provide feedback on inadequate modularization and level of abstraction:

- Having many global flows suggest a poorly refined data structure. We may need to segment a global variable into several pieces.
- Having a high module complexity suggests an improper modularization. It is desirable that a procedure be in one and only one module.
- Having many global flows and a low module complexity suggests that many procedures have access to the global variable, but there is little communication among them.

- Having few global flows and a high module complexity suggest a poor functional decomposition within the module or a complicated interface with other modules.

Henry and Kafura validated their metric by evaluating 10 Unix modules containing 220 procedures. They showed that the occurrence of program changes strongly correlates with their metric. Some results are:

- 72% of procedures which are in more than one module required changes; in contrast with 38% of procedures in only one or less than one (not directly reference and global variable) module required changes.
- 92% of procedures with complexity 10^5 required changes.

They consider four factors: length, length^2, fan-in×fan-out, (fan-in×fan-out)^2, and determine that the last contributes most to the complexity correlations (correlation coefficient of 0.98 with occurrence of code changes).

Yau and Collofello's Design Stability Metric

Yau and Collofello's design stability metric [20] is based on the theory of data abstraction and information hiding. Inadequate abstraction can result in procedures making many "assumptions" about other procedures. If a change is made that affects these assumptions, a "ripple effect" may occur throughout the system, requiring additional change. They consider two simple kinds of assumptions: those about the basic types of entities (such as integer, real, Boolean, character), and those about the values of the basic entities.

Identifying the assumptions associated with a procedure's interface (its parameters and global variables it uses) involves examining each parameter and global variable to see if it is composed of other identifiable basic entities. For each procedure i one identifies

M_i = {procedures invoking i},
M_i' = {procedures invoked by i},
P_{ij} = {parameters returned from i to procedure j, where $j \in Mi$},
P_{ij}' = {parameters passed from i to procedure j, where $j \in Mi'$}.
GR_i = {global variables referenced in i}
GD_i = {global variables defined in i}[1]

For each global variable x, one identifies

$$G_x = \{i | x \in (GR_i \cup GD_i)\}$$

[1]Yau and Collofello say "global variable defined in a module" but the rest of their prose speaks of "modules invoking modules", suggesting their notion of "module" is close to ours of "procedure."

The steps in the computation are then:

1. For each set P_{ij} and each parameter $x \in P_{ij}$, find the number of assumptions made by procedure j about x using the following pseudocode algorithm.

   (a) If parameter x is a structured item, then (recursively) decompose x into its base types and increment the assumption count by 1 (each time).

   (b) For each basic entity comprising x, if procedure j makes assumptions about the values which the basic entity may assume, then increment the assumption count by 2, else increment the assumption count by 1.

   Set TP_{ij} equal to the total number of assumptions made by j about the parameters in P_{ij}.

2. For each set P'_{ij} and each parameter $x \in P'_{ij}$, find the number of assumptions made by procedure j about x using the pseudocode algorithm in step 1. Set TP'_{ij} equal to the total number of assumptions made by j about the parameters in P'_{ij}.

3. For each procedure i and every global variable $x \in GD_i$, find the number of assumptions other procedures make about x. This requires using the set G_x and applying the algorithm in step 1 for each global variable x and every procedure $j \in (G_x - \{i\})$. For each global variable in GD_i, set TGD_i equal to the total number of assumptions made about it by other procedures.

Then, for each procedure i, compute the design logical ripple effect (DLRE) and design stability (DS) as:

$$\text{DLRE}_i = TGD_i + \sum_{j \in M_i} TP_{ij} + \sum_{j \in M'_i} TP'_{ij}$$

$$\text{DS}_i = 1/(1 + \text{DLRE}_i).$$

Then the system design stability (SDS) is:

$$\text{SDS} = 1/(1 + \sum_i \text{DLRE}_i)$$

This metric considers a design's resistance to change. In a poor design a simple maintenance change will ripple through many procedures. Conversely, a good design will contain the change within a single procedure.

Six graduate teams used Structured Design to produce a complete system design specification (estimated 4K lines of program). Proposals for change were analyzed on their potential ripple effect. Several interesting results are reported:

- The procedures that would have contributed large ripple effects if modified are among the procedures possessing poor design stability measure. However, the converse is not necessarily true.

- Many of the procedures found to possess poor stability also are of weak functional strength, and are common coupled to many other procedures.
- The degree of procedure fan-in and fan-out does not always correlate with design stability.

Despite these interesting results, the authors do not consider their metric validated. They did not perform a formal experiment, since that would require a large use-oriented maintenance database.

Card and Agresti's System Complexity Metric

Card and Agresti [3, 5] distinguish between the complexity of connections between procedures (structural complexity) and the internal structure of each procedure (local complexity). To them, the most basic relationship is that a procedure may call or be called by another procedure. They measure the complexity contribution of these calls by counting occurrences of fan-in and fan-out (calls to and from a given procedure, respectively). In the data they have analyzed, multiple fan-in (the calling of a procedure from several places) is generally confined to procedures that perform simple mathematical functions reused throughout the system. Consequently, high fan-in is not an important complexity measure. On the other hand, fan-out is more important. They also note that it is not just total fan-out, but also the distribution of fan-out within a system that affects complexity. They propose the following formula for structural complexity:

$$S = (\sum_{i=1}^{n} f_i^2)/n$$

where f_i = fan-out of procedure i, and n = number of procedures in system. Fan-out does not include calls to system or standard utility routines, but does include calls to procedures reused from other application programs.

For a procedure Card and Agresti believe the workload mainly consists of data items that are input to or output from higher or parallel procedures. The *local complexity* of a procedure is therefore directly dependent on its own "I/O" complexity. The local complexity of procedure i, L_i, is:

$$L_i = v_i/(f_i + 1),$$

where v_i = number of "I/O" variables used by procedure i; "I/O" variables include distinct parameters in the calling sequence (an array counts as one variable) as well as referenced COMMON (Fortran is used) variables. Since Card and Agresti do not consider fan-in in their metric, they use "+1" term in this measure to represent the subject procedure's share of the workload.

Assuming that the workload of a procedure is evenly divided among itself and subordinate procedures leads to the following formula of local complexity L:

$$L = (\sum_{i=1}^{n} L_i)/n.$$

Since all complexity resides in one or the other of the structural complexity and local complexity, the total *system complexity* C of a design is the sum of structural plus local complexity: $C = S + L$.

The design complexity can be minimized by minimizing its structural and local complexities. Minimizing structural complexity requires minimizing the fan-out from each procedure. On the other hand, local complexity can be minimized by minimizing number of variables or maximizing fan-out – this means that fan-out contributes both positively and negatively.

Card's group has compared the computed complexity scores (S, L, C) with a subjective rating of design quality for eight flight dynamics systems. A senior manager who participated in all eight projects subjectively ranked them by design quality. Then, the four best-rated designs were classified as "good" and the other four as "poor". The four designs subjectively rated as "good" also have the lowest relative complexity. Although the correspondence between subjective design rating and computed design complexity is not one-for-one, the data provides some evidence for a relationship.

Shepperd's IF4 Metric

Shepperd [16, 18] proposed a change to Henry and Kafura's information flow metric. Shepperd restricts his metric to architectural information, namely, procedure calling hierarchy, and global variable access. To understand Shepperd's metric, the following definitions are needed:

- A local flow exists from procedure A to procedure B if A calls B and passes a parameter to B, or from B to A if A calls B and B returns a (result) parameter to A.
- A global flow exists from procedure A to procedure B if A updates a global variable and B retrieves from the variable.
- if4_fanin$_i$ = the number of local and global flows terminating at procedure i with all duplicates removed.
- if4_fanout$_i$ = the number of local and global flows emanating from procedure i with all duplicates removed.

The complexity metric termed *IF*4 modified from Henry and Kafura's metric for procedure i is:

$$IF4_i = (\text{if4\_fanin}_i \times \text{if4\_fanout}_i)^2$$

The if4_fanin is multiplied by the if4_fanout to give the number of information paths through a procedure. The metric for a system with n procedures is:

$$IF4 = \sum_{i=1}^{n} IF4_i$$

Shepperd validated the IF4 metric using teams of second year B.Sc. students at Wolverhampton Polytechnic (U.K.), and found that information flow is closely related to development effort.

Yin and Winchester's Network Metric

Yin and Winchester's network metric [21] is based on the notation of procedure call graph (which they view as basically a tree structure, and divide into hierarchical levels). Yin and Winchester's metric has been validated on some large software system projects at Hughes Aircraft Company.

To understand the metrics, the following notations are needed:

N_i: number of procedures from level 0 to level i.

A_i: number of "network arcs" (calling arcs) from level 0 to level i.

T_i: number of "tree arcs" (eliminating those calling arcs from "network arcs", which emanate from a procedure to another same level procedure or to a higher level procedure) from level 0 to level i. T_i actually equals $N_i - 1$.

$\Delta T_i = T_i - T_{i-1}$

$\Delta A_i = A_i - A_{i-1}$

Yin and Winchester define their C-, R- and D-metrics as following:

$$C_i = A_i - T_i$$

$$R_i = 1 - T_i/A_i = C_i/A_i$$

$$D_i = 1 - \Delta T_i/\Delta A_i$$

C_i is a monotonically non-decreasing function that shows the fluctuation of the increment of T_i against A_i. $C_i = A_i - T_i$ measures the "network complexity". R_i measures the "tree impurity" (deviation from a pure tree structure) of level i against level 0. D_i measures that of level i against level $i-1$. This measure is based on the belief that the more the call graph of a design deviates from a pure tree structure, the worse the design is. Values of R_i and D_i lie between 0 and 1. Both equal zero for a tree structure and increase as the system moves away from a tree structure. A sharp increase of C_i from C_{i-1} reveals an heavily increase of network complexity and suggests a place for change.

Yin and Winchester correlated their C metric against error counts in two software systems. Correlation coefficients between the final C value and the total errors in the first and second systems are 0.98 and 0.99 respectively.

Benyon-Tinker's Software Complexity Metric

Benyon-Tinker's software complexity metric [2] depends on the depth and breadth of the call graph of a system. The measure is:

Code	Type	Explanation
R1	E1 $\leftrightarrow$ E1	a procedure calls another procedure
R2	E1 $\leftrightarrow$ E22	a procedure reads from a global variable
R3	E1 $\leftrightarrow$ E22	a procedure writes to a global variable
R4	E1 $\leftrightarrow$ E21	a parameter returned from a procedure
R5	E1 $\leftrightarrow$ E21	a parameter passed to a procedure
R6	E1 $\leftrightarrow$ E22	a global variable defined in a "module" (procedure)
R7	E2 $\leftrightarrow$ E3	datum has type
R8	E3 $\leftrightarrow$ E3	type is composed of other type(s)

Table 2: Relationships in Metrics Data Model

$$C_\alpha = (\sum_{l=1}^{m} n_l \times l^\alpha) / \sum_{l=1}^{m} n_l,$$

where, n_l is the number of distinct procedures at level l; m is the maximum depth of the call graph; and α is a "power-law" index. Benyon-Tinker believes that an increase of 25% is reasonable when a system evolves from level to level, so recommends an α value of between 2 and 3. The denominator in the metric attempts to remove the effect of system size.

Data Needed to Calculate the Design Metrics

In this section we show an entity-relationship data model for design information, from which some of the metrics can be computed. Our hope is that one could populate a design database with information about a particular design, then compute all the metrics from the database. The entity sets in the model are:

- E1 – procedures, and
- E2 – input/output data, divided into
  - E21 – parameters, and
  - E22 – global variables.
- E3 – types

Table 2 summarizes the relationships (associations among entities). E1 elements have an attribute, computable from the call graph, giving their level in the call hierarchy.

Table 3 shows the elements and relations each design metric needs. For reasons of space it omits E3, R7, and R8, which are used only in Yau and Collofello's metric. The last column summarizes code-level properties each metric requires:

[2]Chapin does not divide E2 into its subsets; all others using E2 require distinguishing parameters from global variables.

Metric	E1	E2[2]	R1	R2	R3	R4	R5	R6	Code
Henry & Kafura	*	*	*	*	*	*	*		1, 2
Yau & Collofello	*	*	*	*	*	*	*	*	3
Card & Agresti	*	*	*	*	*		*		
Shepperd	*	*	*	*	*	*	*		
Yin & Winchester	*		*						
Chapin	*	*	*						4
Benyon-Tinker	*		*						

Table 3: Metrics & Data Cross Reference

1. Length.
2. Information returned from one procedure and passed to another.
3. Assumptions made about values of parameters.
4. Variables used in determining control flow.

We omit McClure's metric, which does not fit our initial data model. Even the metrics that do not require code-level information do require quite detailed information about the entities, including the call graph, variable usage, and parameters. Thus all are computable only at a fairly late stage of design.

Evaluation of Design Metrics

This section summarizes some advantages and disadvantages of each metric.

Henry and Kafura's Information Flow Metrics

Henry and Kafura's information flow metric helps in locating potential design and implementation problems. The idea behind this metric is that the more complex procedures in a system are those through which large amounts of information flow. It may be used to identify potential problem modules and procedures by concentrating on those with abnormally high complexities.

Henry and Kafura have applied their metric to the Unix operating system and have had some success in identifying problematic areas. They have also found a high correlation (r=0.94) between information flow and number of errors (measured as the number of program changes).

Although Henry and Kafura's metric has been validated by some researchers [8, 10], there are still some points to be further considered:

- The square in their metric seems subjective.

- The metric makes simple assumptions about information; it neither distinguishes between control and data flows, nor between simple and complex data structures.

Yau and Collofello's Design Stability Metric

Design stability requires a deeper analysis of procedure interfaces than do other design metrics. The use of assumptions about procedure interfaces is also more accurate that measures such as procedure fan-in and fan-out (which only examine invocations), or procedure coupling measures. However, using this metric requires structural knowledge of internal design details.

Card and Agresti's System Complexity Metric

Card and Agresti identify total design complexity as comprising of structural complexity plus the sum of individual procedure fan-outs; thus procedures with many subordinates will score highly. They disregard fan-in as their work has shown it to be insignificant [4]. Local complexity for a procedure is the number of imported and exported variables, divided by the fan-out plus one. The rationale for this, is that the greater the number of arguments, the greater the procedure workload. On the other hand, the greater the procedure fan-out the greater the proportion of this workload that is distributed to other procedures.

For Card and Agresti's system design complexity metric, two aspects of the current complexity measurement need to be considered:

- All validated systems were written in FORTRAN, and mainly deal with numerical data processing.
- An extended application of the metric to design using different formalisms intended using different implementation languages is expected.

Additionally, some important design practices (e.g. information hiding and data abstraction) have been excluded from Card and Agresti's empirical validation, because they are difficult to measure or implement in FORTRAN.

Shepperd's IF4 Metric

There are some major differences from Henry and Kafura's metric:

- The simple length measure is omitted.
- Only are those indirect local flows (as defined in Henry and Kafura's metric) counted as local flows in Shepperd's metric, which fall in case 1 of Henry and Kafura's indirect local flows.
- Duplicated flows are ignored.

Yin and Winchester's Network Metric

Yin and Winchester's metric is based on how far the procedure call graph network departs from a pure tree. A validation of C_i against two projects at Hughes Aircraft Company has produced a high positive correlation between their metric and the error count. However, as Yin and Winchester note, this is in part because of the good partial correlation of some large components and the dispersion of these points results in high correlation of the whole set. Once these points are discarded, they obtain a correlation coefficient of r=0.52.

For Yin and Winchester's metric, one aspect of their work need to be considered: this metric ignores the use of common procedures. Although the designer should seek to minimize C_i where a choice exists, this should not override the reuse of components where possible.

Conclusion

Several sources [9, 17] list the potential applications of design metrics, including their use for prediction, quality control, and decision making (about alternative designs). Software design is a decision making process. Decisions based on design evaluation allow to judge that one design is preferable to another. Metrics can be used to provide a basis to identify outlier procedures, measure design attributes, and help to generate new solutions, thus improving design quality.

Although several metrics have been proposed, none of them has been fully validated or shown to be without flaws. Our work suggest that variants of most of them can be computed from a common data model; we expect to continue working on establish a common basis for computing them.

Acknowledgements

Tom Dean, Andrew Malton, and Kevin Schneider gave helpful comments on earlier drafts of this paper. This work was supported in part by the Information Technology Research Centre of Ontario, and in part by the Natural Sciences and Engineering Research Council of Canada.

References

[1] C. Alexander. *Notes on the Synthesis of Form.* Harvard University Press, Cambridge, MA, USA, 1964.

[2] G. Benyon-Tinker. Complexity measures in an evolving large system. In *Proc. Workshop on Quantitative Software Models*, pages 117–127, Kiameshalake, NY, USA, October 1979.

[3] D. N. Card and W. W. Agresti. Measuring software design complexity. *Journal of Systems and Software*, 8:185–197, 1988.

[4] D. N. Card, V. E. Church, and W. W. Agresti. An empirical study of software design practices. *IEEE Transactions on Software Engineering*, 12(2):264–271, February 1986.

[5] D. N. Card and R. L. Glass. *Measuring Software Design Quality.* Prentice-Hall, 1990.

[6] N. Chapin. A measure of software complexity. In *Proc. National Computer Conference*, pages 995–1002, 1979.

[7] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE-7(5):510–518, 1981.

[8] S. Henry, D. Kafura, and K. Harris. On the relationship among three software metrics. In *SIGMETRICS Performance Evaluation Review*, pages 81–88, 1981.

[9] D. Ince. Software metrics: Introduction. *Information and Software Technology*, 32(4):297–303, May 1990.

[10] D. Kafura and J. Canning. A validation of software metrics using many metrics and two resources. In *Proc. 8th International Conference on Software Engineering*, pages 378–385, London, UK, 1985.

[11] D. Kafura and S. Henry. Software quality metrics based on interconnectivity. *Journal of Systems and Software*, 2:121–131, 1981.

[12] C. L. McClure. A model for program complexity analysis. In *Proc. 3rd International Conference on Software Engineering*, pages 149–157, May 1978.

[13] D. L. Parnas. On the criteria to be used in decomposing system into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

[14] Arthur Ryman. Foundations of 4thought. In *Proceedings of the 1992 Centre for Advanced Studies Conference (CASCON'92)*, pages 133–155, November 1992.

[15] M. Shepperd. An evaluation of software product metrics. *Journal of Information and Software Technology*, 30(3):177–188, April 1988.

[16] M. Shepperd. A metrics based tool for software design. In *Proc. Software Engineering '88*, pages 45–49, 1988. Second IEE/BCS Conference.

[17] M. Shepperd. Metrics, outlier analysis and the software design process. *Information and Software Technology*, 31(2):91–98, March 1989.

[18] M. Shepperd. Design metrics: An empirical analysis. *Software Engineering Journal*, 5(1):3–10, January 1990.

[19] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, 1974.

[20] S. S. Yau and J. S. Collofello. Design stability measures for software maintenance. *IEEE Transactions on Software Engineering*, SE-11(9):849–856, September 1985.

[21] B. H. Yin and W. Winchester. The establishment and use of measures to evaluate the quality of software design. In *Proc. Software Quality and Assurance Workshop*, pages 45–52, 1978.