THE USE OF SOFTWARE QUALITY METRICS IN SOFTWARE MAINTENANCE

by

Dr. Dennis Kafura
Mr. Geereddy R. Reddy

TR-85-33

August 1985

THE USE OF SOFTWARE COMPLEXITY METRICS IN SOFTWARE MAINTENANCE

Dr. Dennis Kafura
Mr. Geereddy R. Reddy

Department of Computer Science
Virginia Polytechnic Institute
Blacksburg, VA 24061

Abstract

This paper reports on a modest study which relates seven different software complexity metrics to the experience of maintenance activities performed on a medium size software system. Three different versions of the system that evolved over a period of three years were analyzed in this study. A major revision of the system, while still in its design phase, was also analyzed.

The results of this study indicate: (1) that the growth in system complexity as determined by the software metrics agree with the general character of the maintenance tasks performed in successive versions; (2) the metrics were able to identify the improper integration of functional enhancements made to the system.; (3) the complexity values of the system components as indicated by the metrics conform well to an understanding of the system by people familiar with the system.; (4) an analysis of the redesigned version of the system showed the usefulness of software metrics in the (re)design phase by revealing a poorly structured component of the system.

This work was supported, in part, by grants from the National Science Foundation (MCS-8103707, DCR-8207110, DCR-8418257).

I. Introduction

An important concern for software engineers today is the rising costs of maintaining software systems. Maintenance, in the sense used in this paper, encompasses not only the repair of errors but also includes enhancements to the system's operation. The costs for such software maintenance activities have been observed to outweigh the development costs [5] and take a greater share of the total software budget for many organizations than development costs. The high cost of maintenance can, in part, be attributed to the greater difficulty in controlling the maintenance process than processes in other phases of the life cycle. In comparison with these other phases, a more diverse group of people over a longer period of time affect the system being maintained. Furthermore, this diverse group is usually not coordinated by a common "maintenance method". While numerous methodologies exist for requirements analysis, design (both high- and low-level), and testing, no comparable cohesive force binds together the separate activities of maintainers working independently and at different times. This disparate maintenance activity leads to system whose complexity grows rapidly with time [3] and which are slowly, but unavoidably, maintained to death [6].

In this paper we present encouraging results from a very limited study. This study explores the relationships between a variety of software complexity metrics and the effects of maintenance activities on a *single* medium-size system. These results suggest that the quantitative information provided by the software complexity metrics could be used to form the control element in a complete maintenance method. For example, software complexity metrics computed from the complete source code or a design representation of planned enhancements could be used to assess the impact of maintenance activities on a system's structural complexity or to help judge whether enhancements have been inserted into the proper place in the system's structure.

The relationship between metrics and maintenance may be explored using either objective or subjective techniques. An objective study typically performs a correlation analysis between the complexity metrics of a given component and the times to perform maintenance tasks on that component. There are numerous examples where quantitative techniques have been used to investigate the development phase of the life cycle (see, for example, [2] [7]). A recent study of the maintenance phase was conducted by Rombach [20] [21] in which he found a significant relationship between quantitative measures similar to those used in this paper and assigned maintenance tasks.

In this study we use a subjective evaluation technique. This means that we will attempt to relate the quantitative measures defined by the software metrics to the informed judgement of experts who are intimately familiar with the system being studied. The use of a subjective evaluation technique was motivated primarily by our desire to see if software metrics could provide a maintainer, or a maintenance manager, with information that was consistent with expert experience and which could be used as a guide to avoid poorly performed maintenance. This subjective evaluation, of course, must be credible to the the reader and must be consistent with accepted concepts of software engineering. The subjective evaluation technique has been used previously with good success to study design issues [13]. Necessity also urged us in the direction of a subjective experiment since objective historical data on the maintenance of the system being studied was not kept. These records were not collected because the study being reported in this paper was conducted in retrospect; it was not an anticipated part of the system's development or goals. Finally, we believed that the combination of objective and subjective experiments presents a more compelling case than either form of experiment by itself. Since considerable work had already been done involving an objective approach we decided to pursue the subjective approach.

The next section of this paper explains in more detail the elements which are part of the study: the complexity metrics which were used, the system whose maintenance was being observed, and the qualifications of the experts used to evaluate the metric information. Section III reports the results of the longitudinal study of the MDB system. This approach is similar to the work of Belady and Lehman [3] [17] differing from their work in that we observe the system in more detail but over a shorter period of time. As in the Belady and Lehmann study, the growth of system complexity over time is a major factor of interest. Changes in the metrics at the system level and metrics of individual components are considered. Section IV examines the "outlier" components in one version of the system. These components exhibit the most extreme measure on one or more metric scales. In studies of the development process, such "outliers" have been found to be associated with anomalously high error rates and coding times [16] [22]. Section V briefly states the conclusions that can be drawn from this study.

II. The Metrics, System, and Experts

This section describes the different complexity measures that were used, the system being studied, and the qualifications of the "experts" used to assess the information provided by the metrics.

There are seven different metrics used in this study. These metrics are:

1. McCabe's Cyclomatic complexity number [18]
2. Halstead's Effort metric E [9]
3. Lines of code
4. Henry and Kafura's Information Flow Metric [11]
5. McClure's Control Flow Metric [19]
6. Woodfield's Syntactic Interconnection Measure [23]
7. Yau and Collofello's Logical Stability Metric [24]

These complexity metrics can be divided into two classes, code metrics (metrics 1-3 above) and structure metrics (metrics 4-7 above). These seven metrics (and other variations on these metrics) were generated by a sophisticated analysis tool. The analyzer was an enhancement of an information flow analyzer developed under the direction of Dr. Sallie Henry. The information flow analyzer performs a lexical and semantic analysis of FORTRAN source programs and generates the first four metrics listed above. As part of this study and also [7], the analyzer was modified to incorporate the remaining three metrics. Metrics from these two classes were used because of the apparent difference between the factors measured by metrics in these classes. [15] [12].

The code metrics focus on the individual system components (procedures and modules) and require a detailed knowledge of their internal mechanisms. One of the reasons these types of measures have attracted such attention is the relative ease with which they can be calculated. In contrast to the structure metrics described later, only the simplest of tools is needed to determine the McCabe metric for a structured program. McCabe cites a close correlation between an objective complexity ranking of 24 in--house procedures and a corresponding subjective reliability ranking by project members [18]. McCabe's metric and Halstead's effort metric were also studied by Curtis [8]. Basili studied these and other metrics utilizing data extracted from the Software Engineering Laboratory at NASA Goddard Space Flight Center [1].

Structure metrics on the other hand view the product as a component of a larger system and focus on the interconnections of the system components. Henry and Kafura's information flow metric was used in an objective study of the UNIX operating system [10] . This study found a statistical correlation of 0.95 between errors and procedure complexity as measured by the

information flow metric. Although similar correlations for McCabe's metric (0.96) and Halstead's effort metric (0.89) were found, it was observed that the code metrics were highly correlated to each other but only weakly related to the information flow metric. This result suggested that the information flow metric measured a dimension of complexity different from the other two metrics.

McClure's metric focuses on the complexity associated with the control structures and control variables used to direct procedure invocation in a program. McClure argues that all predicates do not contribute the same complexity. She associates a higher complexity with those control variables appearing in conditional statements which determine the invocation of other procedures. The complexity of a program module P consists of two factors: first, the complexity associated with the control variables invoking module P and second, the complexity associated with the control variables by which module P invokes other modules. The overall complexity is calculated by adding together the complexities of the modules. McClure makes two recommendations with respect to the complexity of a partitioning scheme: the complexity of each module should be minimized and the complexity among modules should be evenly distributed.

Woodfield's comparison of code metrics led to the development of a hybrid model that includes module interconnections. He found that the software science effort measure when combined with a model of programming based on logical modules and module interconnections produced the closest estimates to actual programming times [23]. Woodfield tested his metric on data collected from student programmers developing programs for a programming competition. There were thirty small programs (18 - 196 lines of code) and the time needed to complete each program was compared with results obtained by using the predictions from his metric. His results indicated that the model was able to account for 80 percent of the variance in programming time with an average relative error of only 1 percent.

Yau and Collofello present a measure for estimating the "stability" of a program [24], which indicates the resistance to the potential ripple effect observed when a program is modified. A primitive maintenance activity is utilized in measuring the stability of a program. This primitive activity is a change to a single variable definition in a module. The authors justify this by saying that regardless of the complexity of the maintenance activity, it basically consists of modifications to variables in a module. Their "logical ripple effect" represents a measure of the expected impact on the system of a modification to a variable in a module. A measure for the stability of a module is defined as the inverse of this ripple effect measure.

The system being studied in this paper is a data base management system called the Mini Data Base (hereafter referred to as MDB). It is based on a relational model and it runs under the VMS operating system on a VAX 11/780. The MDB system is written in FORTRAN and supports only a single user. Each user process has a complete copy of the MDB executable code and has exclusive use of one database file at the time it is in use. The MDB provides a variety of commands which enable users to create a database, define, load, store or drop relations in a database, load and store tuples into and from a relation, insert, modify and delete individual tuples in a relation, query relations, rerun commands or use an editor to modify them, and execute files of commands. The MDB is a medium size software system (16,000 lines of FORTRAN code) developed by graduate students of the Computer Science Department at Virginia Tech over the last 7 years. Mr. Reddy himself worked with this system in different roles and is quite familiar with its design and implementation.

While MDB is a "student" system we believe it is a realistic object of study from which to learn about software maintenance. First, the MDB is a fully functional database system. It is being used as a support tool in other research projects managing real data. Second, the maintenance tasks included both repairs of errors as well as the addition of major functional enhancements. Third, more than 100 different people contributed to the various maintenance tasks over the lifetime of the MDB under the direction of a project administrator. Fourth, the individuals involved in the project

were at least second year Masters degree candidates, many of whom were subsequently hired as "professional" programmers.

A brief overview of the history of the MDB system is given in order to familiarize the reader with the system being studied in this paper and also to introduce several key components of the system which figured prominently in the maintenance of the MDB system.

The original work on MDB started in the spring of 1977 as a project in the Department of Computer Science at Virginia Tech. A graduate level course in database systems formulated the goals and the design details of a database management system for an LSI-11 microcomputer. A second database class in the spring of 1978 produced a working system - a limited DBMS using floppy disk storage. This was based on the original goals of the previous year and on revisions of that design. Over the next few years, students worked on parts of the MDB for their Masters projects and the system was moved to the VAX-11/780. Starting in 1981 the following versions of the MDB system have been produced.

Version 1 (Spring 1981)

This is the first of three versions used in this study and is also the oldest version for which the complete source code was available. This version has most of the functions of the final MDB except for sorting and merging. In addition, other functions - like the data definition language and session management - also existed, but only to a very limited extent.

Version 2 (Spring 1982)

Many enhancements were made to the MDB in this version. Almost all of these enhancements were new commands and new capabilities. Some of the new features include the capabilities to store and load data, schema, and command files, to invoke execution of command files, to modify a previous command using an editor, to run VAX DCL commands without leaving the MDB environment, to save all changes to the database automatically or on demand, and enhance the capabilities of a SELECT command used to query the database for tuples.

One interesting aspect of the enhancements made in this release is that 12 new commands were added. The code for recognizing and processing these new commands was added in an improper place. The reasons behind the improper placement of these enhancements will be explored later in the paper.

Version 3 (Spring 1983)

Most of the enhancements made in this release are in the form of bug fixes. The only major improvement was in the sorting and merging function. A few cosmetic changes were also made in displaying the schemas.

Version 4 (Spring 1984)

During the Fall of 1983, an advanced graduate class in Information Systems started on a project to redesign the MDB. That work was followed up by an other class in Winter 1984 to do more detailed design work and the required programming of the new MDB. Although, a complete source code was not available at the time of this study, the design was specified to an extent that identifies the different procedures and their various interconnections. The level of detail was sufficient to allow the analyzer to compute the structure metrics for this version.

Two experts on the MDB system were asked to participate in the interpretation and assessment of the software metrics. One of them, Dr. Rex Hartson, has been the director of the MDB project since its inception. Since he worked with all versions of the MDB that evolved over the last 7 years, he is very familiar with the MDB functions and their evolution. The other person, Mr. Bob Larson worked with all the versions of the MDB analyzed in this study including the New MDB. He served as the project administrator for the New MDB project. These two are easily the two persons most familiar with the MDB design and implementation.

III. Analysis of Complexity Changes

One important part of this study was to analyze the change in complexity of the MDB from one version to the next. The comparisons between the three versions of the MDB were made at two different levels of the system. First, the total complexity of the system for each of the seven metrics was computed for each version and these complexity values compared. Second, the percent change in complexity from one version to the next for each procedure was computed and the results analyzed. The following subsections discuss the results obtained by using these two approaches.

Complexity Change - System Level

For each of the seven metrics used, the total complexity at the system level was computed by summing up the complexities of the individual procedures in the system. This was done individually for each of fthe three MDB versions. The results obtained are shown in Figures 1, 2, and 3. The complexity values on the vertical axis for each metric were normalized by dividing the complexity value of each metric by the Version 1 complexity for the same metric. Thus, a complexity value of 1.6 for the lines of code metric for Version 2 implies that the increase in system complexity from Version 1 to Version 2 was 60% as measured by the lines of code metric. Figure 1 shows the graph of the increase in system complexity for the code metrics. Figures 2 and 3 show, respectively, the corresponding data for the information flow metric and the three other structure metrics.

As can be seen from Figures 1-3, the increase in the total system complexity from Version 1 to Version 2 is significantly larger than the increase in complexity from Version 2 to Version 3. As may be recalled from the previous section, many new enhancements were made to the MDB in Version 2 and almost all of these enhancements consisted of adding new commands and new capabilities to the MDB. On the other hand, most of the maintenance performed on Version 3 were error correction. Thus, the change in complexities over time agrees with what one would expect. That is, the simple repair of errors should introduce proportionately less change in the system's structure and coding than should changes and additions made to incorporate significant functional enhancements.

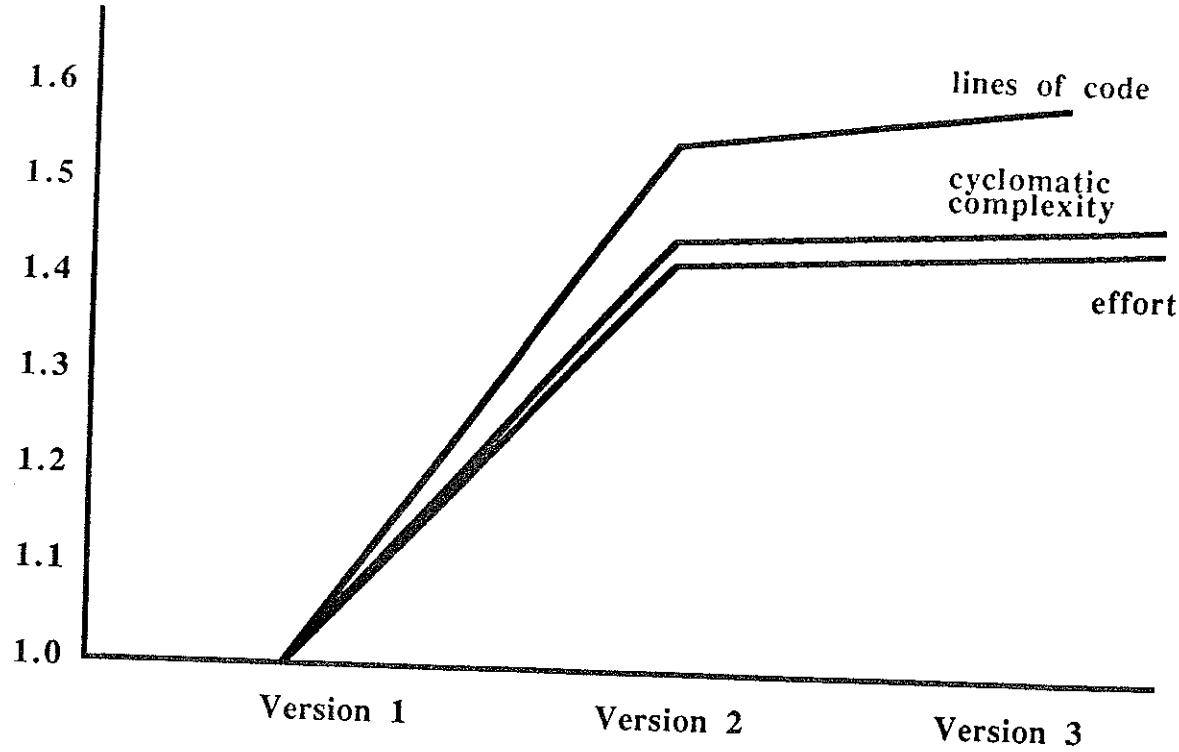

Figure 1: System Level Complexity Increases for Code Metrics

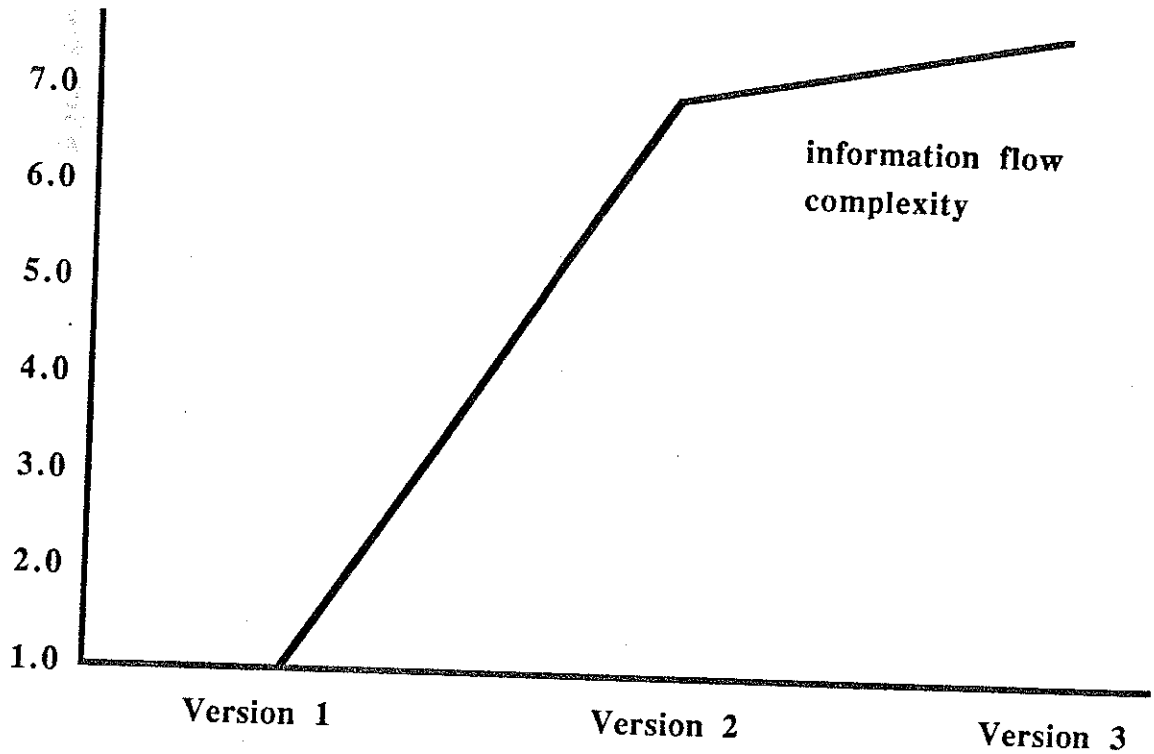

Figure 2: System Level Complexity Increases for Information Flow metric.

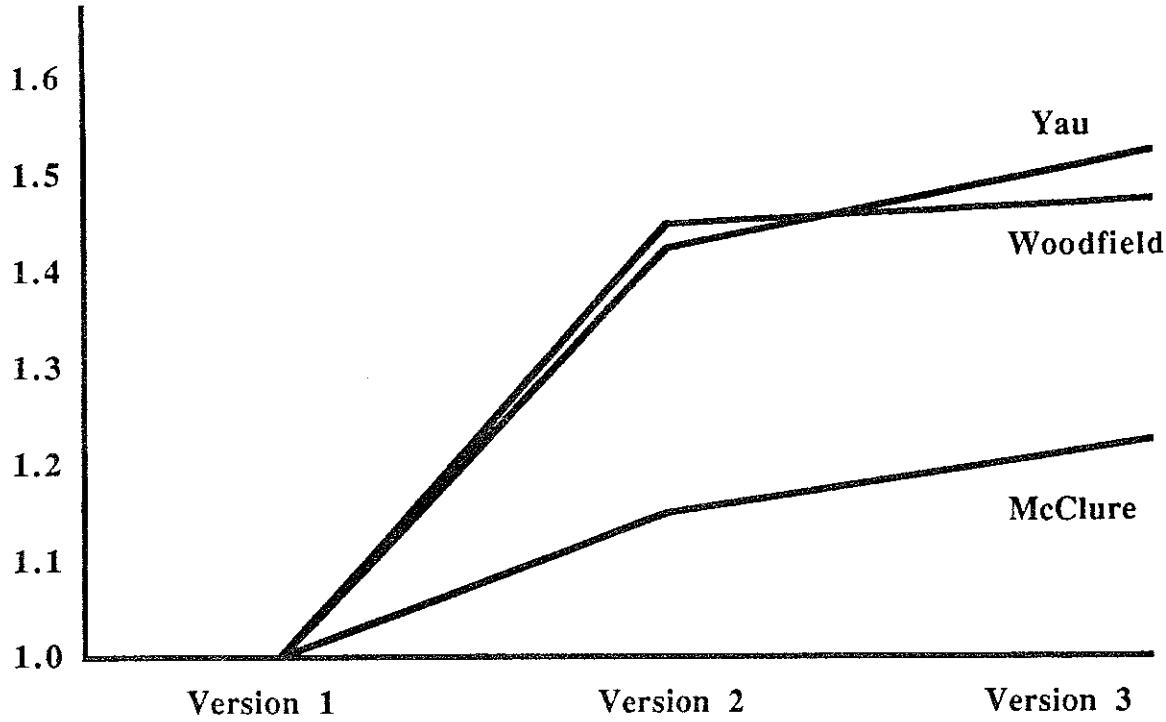

Figure 3: System Level Complexity Increases for structure metrics

Two additional observations should be made about Figures 1-3. First, no significance should necessarily be attached to the larger scale which appears in the information flow complexity graph (Figure 2). The information flow metric tends to magnify changes more than other metrics because it involves a quadratic term. Second, it is interesting to note that the complexity growth curves for all of the metrics give evidence of the same overall trend. This seems to support the view that maintenance activities - either enhancements or repairs - impact many different aspects of the system simultaneously. We do not see, for example, that the repairs were accomplished by only localized changes which did not affect the global structure of the system. Quite the contrary, the repairs caused increases in both the code metrics and the structure metrics. While it is certainly true that cases exist where an individual repair or enhancement can be made purely by a localized change to one component, it would appear from our study that the combined effects of numerous changes is not localized. In particular, the growth in structural complexity as a result of maintenance activity is consistent with the Belady and Lehman study [3] and the general perception that systems become more difficult to maintain over time because they become increasingly complex [17].

The data presented in Figures 1-3 confirm that the metrics appear to accurately reflect the growth in complexity of the system introduced as a result of maintenance activities. However, such data does not, by itself, reveal any possible flaws in the way the maintenance to the system was performed. To study how well the enhancements were integrated into the system we need to look at a more detailed level. This is done in the next subsection.

Complexity Change - Procedure Level

The first set of procedure complexity increases we examined were for those between Version1 and Version 2. When the procedures were sorted by the percent increase in complexity, one procedure had the highest increase in complexity for almost all of the seven complexity measures. This procedure was named NEXTQB. Table 1 gives a list of procedures that had very high increases in complexity for at least one, and usually several, of the metrics. As can be seen in Table 1, NEXTQB was the procedure whose complexity was impacted the most by the enhancements made in Version 2. In order to understand the reason for this dramatic increase in the complexity of NEXTQB a detailed analysis of the function performed by NEXTQB in each version was done.

	code metrics			structure metrics			
procedure	**lines code**	**effort**	**cyclo. compl.**	**McClure**	**info. flow**	**Wood field**	**Yau**
Nextqb	468	330	450	1387	4×10^7	445	999
Evalu8	118	86	115	0	118	86	0
Page	-8	-7	0	-47	154	-7	470
Dbmain	62	44	37	-28	2900	83	175
Messg	44	34	0	8	1500	34	0
Stpcmd	70	69	0	79	2600	69	-80

Table 1: Percent Complexity Increase (Version 1 to Version 2)

The procedure NEXTQB as it existed in Version 1 was used to obtain the next query buffer (a line of user input) . As shown in Figure 4, NEXTQB was invoked by the procedure DOLEX which was the driver routine for all lexical analysis in the MDB. Whereas the function performed by NEXTQB in Version 1 was elementary, the DOLEX routine performed a complicated lexical scanning based on one large state table. Furthermore, NEXTQB did not directly interact with many other procedures in Version 1. Therefore, as might be expected, NEXTQB had low complexity values in Version 1 while those of DOLEX were higher.

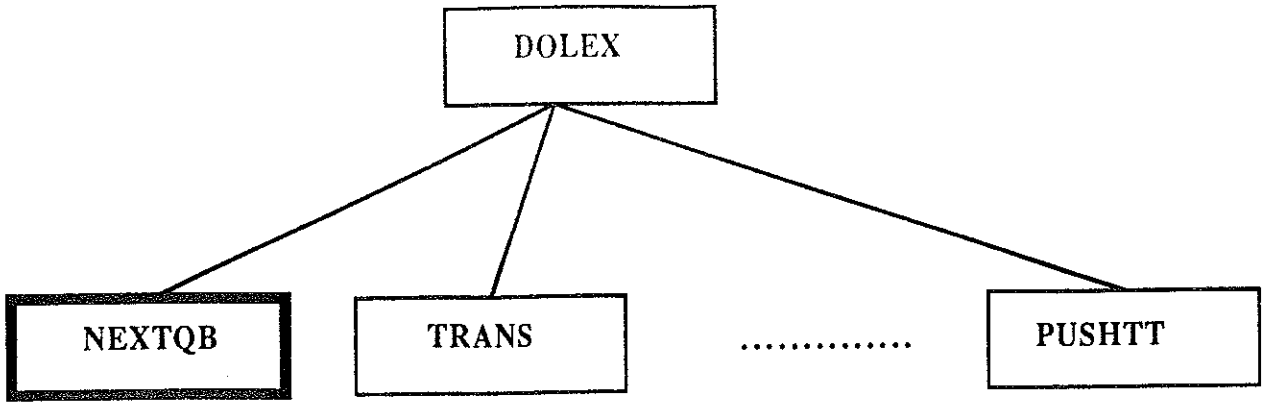

Figure 4: Lexical Analysis in Version 1 of MDB

One of the major enhancements in Version 2 involved the addition of 12 new commands. With each new command an appropriate enhancement had to be made to the system's lexical analysis and semantic processing capabilities to properly recognize and interpret the new commands. One would expect that the driver routine, DOLEX, would be modified to accommodate these new commands - possibly by the addition of new subordinate procedures. Instead, these 12 new commands were recognized in NEXTQB itself and furthermore the routines to execute these new commands were called directly by NEXTQB as shown in Figure 5.

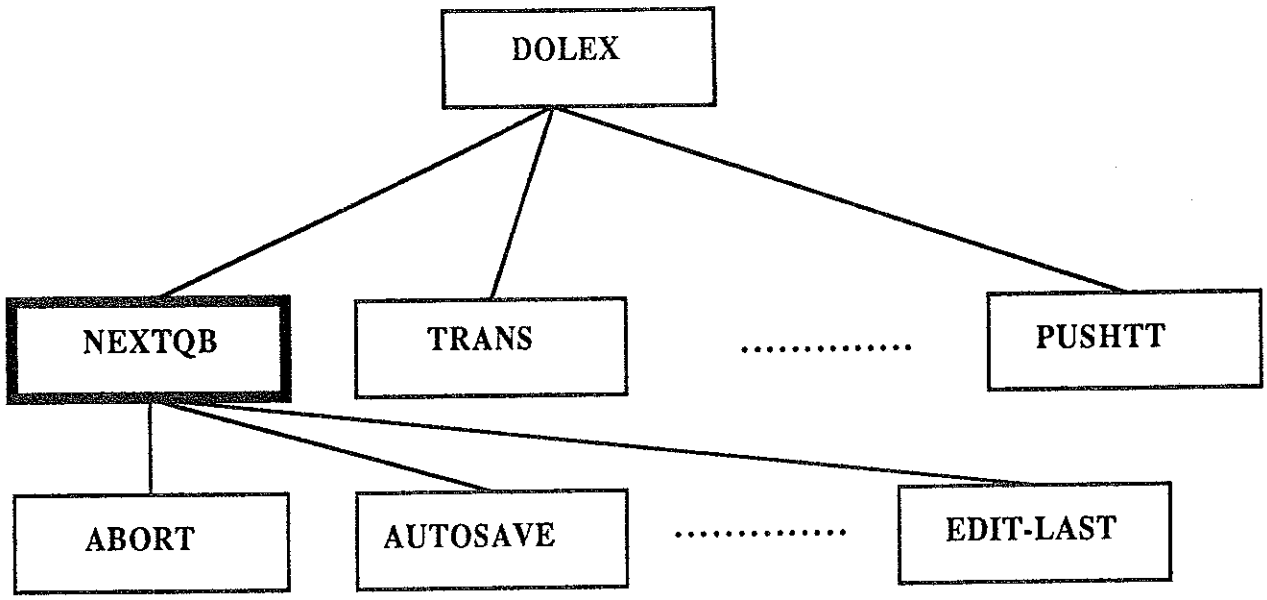

Figure 5: Lexical Analysis in Version 2 of MDB

By any reasonable standards of design and maintenance, the NEXTQB procedure was obviously not the appropriate place to insert the lexical and semantic processing of the the new commands. In effect, a simple routine to obtain the next buffer of user input has now assumed a part of the functional role of both the lexical and semantic processing of the system. In discussions with the MDB project administrators it was found that DOLEX was perceived by the student maintainers to be more difficult to modify than was NEXTQB. In order to properly incorporate the new commands in DOLEX the maintainer would need to understand the current state table arrangement and the code which used this table to drive the command recognition algorithm. By incorporating the changes entirely with NEXTQB the maintainer was able to insert a new self-contained subtree on to the NEXTQB procedure and avoid the expense of learning the DOLEX algorithm.

This example suggests two possible uses for software metrics applied during the maintenance process. First, the metrics can be used to identify improper integration of enhancements. Reviewers and managers can monitor the changes in procedure complexity and question large changes in procedures which are peripheral to the major purpose of the enhancement. Second, procedures which are perceived to be complex, such as DOLEX, can lead to future improper structuring of the system because maintainers will avoid dealing with this complex procedure when making enhancements, even when the maintainer knows that a major restructuring of the complex component is called for in order to gracefully include the required enhancements. It is only natural that, left unchecked, maintainers will choose to reduce their own investment of time to perform some maintenance act even if their approach will lead to a degradation of the structure of the system. Gradual degradations of this kind ultimately produce unmaintainable systems.

The procedure complexity increases from Version 2 to Version 3 were also studied. Table 2 gives the percent increases in complexities for the procedures with the largest increases. As mentioned earlier, most of the maintenance changes made in Version 3 were bug fixes. The relatively smaller complexity increases observed at the system level between Version 2 and Version 3 is also reflected in the relatively lower percentages of the procedure level complexities. This can be seen by comparing the percentage increases in Table 1 with those in Table 2.

	code metrics			structure metrics			
procedure	lines code	effort	cyclo. compl.	McClure	info. flow	Wood field	Yau
Sort	516	297	150	6100	1287	297	33
Switch	0	0	0	0	800	0	75
Schdsd	0	0	0	999	671	14	50
Help	182	6	0	0	182	6	0
Evalu8	17	8	12	0	17	9	0

Table 2: Percent Complexity Increase (Version 2 to Version 3)

One function that was enhanced between Version 2 and Version 3 was the Sort/Merge function. Not surprisingly, the procedures that had the highest increase in complexity were, in general, the one in the Sort/Merge module. For example, the routines SORT and SWITCH form a major part of the Sort/Merge function.

One other interesting observation which can be made on the data in Table 2 bears on the distinction between code and structure metrics. Although the procedures HELP and EVALU8 show a reasonable increase in the code measures, there is little or no change in any of the structure metrics for these procedures. The increase in the structure metrics that does occur is explained by the use of "weighting terms" in some of the structure metrics. In Woodfield's measurement, for example, the complexity of a component is determined by multiplying two terms: one term depends on the control and data connections which this component has with other components and the second term is Halstead's effort measure (E). Thus, the structure complexity can increase due solely to changes in the internal structure of the component. The growth in the information flow and Woodfield's measurements for HELP and EVALU8 are due only to changes in the code metrics. On the other hand, the procedures SCHDSD and SWITCH indicate no change in the code measures while recording significant increases in their structure measures. These changes are indicative of the distinction between code and structure metrics made by Henry and Kafura [14] and elaborated in [15]. It is not a question of whether one class of metrics is better than the other. It is more a question of learning what factors are being measured by metrics in these classes and how to use the combined information which each class offers.

The reader should not confuse two different observations which have been made at this point. It has just been observed that maintenance performed on *individual* components may cause changes in one class of metrics, either code or structure, without causing changes in metrics of the other class. This is not inconsistent with the earlier observation that maintenance performed on a *collection* of components tends to show increases in all metrics.

IV. A Closer Look At Outliers

An analysis of the procedures that have a very high complexity has proven useful in two previous experiments [16] [22]. In these previous studies components with unusually high metric values, termed "outliers", were found to be sources of design, coding, or maintenance problems.

To study such outliers, the Spring 1981 version was chosen for analysis. The outliers were identified by first sorting the procedures by their complexity value for each of the complexity metrics and then taking the union of those procedures that appeared as the most complex procedures for each of the complexity metrics. Table 3 shows the seven complexity metrics for most of the outliers. To better interpret the results, Table 4 shows the mean, standard deviation, minimum and the maximum values for all the seven metrics used to obtain the outliers.

The system components identified by the above procedure agreed well with both the authors' subjective evaluation of the MDB and the subjective evaluation of the MDB by the system administrators (Dr. Rex Hartson and Mr. Bob Larson). In their opinion:

> "The complexity measurement results concur completely with our experience and intuition in identifying those parts of the MDB which were most bug-ridden, troublesome and frustrating to the programming teams doing the maintenance and modification."

	code metrics			structure metrics			
procedure	lines code	effort	cyclo. compl.	McClure	info. flow	Wood field	Yau
Proc	478	3042	106	11	4×10^{8}	9119	23
Cmanal	325	2018	93	6	2×10^{7}	5257	77
Dolex	478	2844	138	5	2×10^{8}	8434	27
Messg	451	1934	1	14	4×10^{8}	5802	0
Page	48	295	7	22	1×10^{8}	886	1
Mvc	8	72	3	10	2×10^{6}	216	70
Inavi	164	1405	28	2	9×10^{6}	4052	25

Table 3: Complexity Outliers (Version 1)

	code metrics			structure metrics			
procedure	lines code	effort	cyclo. compl.	McClure	info. flow	Wood field	Yau
mean	43	298	8	1	1×10^{7}	715	15
st. dev.	78	460	18	3	6×10^{7}	1346	20
minimum	3	21	1	0	3	21	0
maximum	478	3042	138	22	4×10^{8}	9199	119

Table 4: Metric Value Statistics for Complexity Outliers

To allow the reader some insight into the reasons underlying the "intuition and experience" of the the system administrators, a brief discussion of three of the procedures in Table 3 is given below.

. PROC

This procedure executes the user commands and is the highest level procedure in the command execution module. There are three classes of user commands: data definition, data manipulation, and session management. A detailed analysis of PROC shows that, instead of recognizing the class of database command and calling an appropriate routine to perform the corresponding function, PROC has separate internal sections for performing a great deal of processing for each command. Although there are lower level procedures to perform the primitives required for each of the database commands, the command execution module can be split into 3 different procedures that perform the corresponding database function.

To give a specific example of the extensive command execution done by PROC, the command to UPDATE some tuples in the database is done as follows in PROC:

1. A procedure (QPROC) is called to retrieve the qualifying tuples.
2. For each qualifying tuple retrieved, the procedure (QSELCT) is called to print out the tuple to the user for inspection.
3. PROC calls appropriate User Communications routines to obtain input from the user, to find out if the tuple really needs to be updated or not.
4. If the response is yes, the tuple is updated in PROC.There is also a dialogue with the user by which PROC determines if it should continue or terminate the UPDATE function.

In the New MDB the functions performed by PROC are no longer contained in one large procedure. Instead, these functions are implemented in different procedures. This breakup of the PROC procedures is additional evidence that the original design of PROC was ill conceived as was indicated by the metrics.

.CMANAL

This procedure is the main procedure in the Command analysis module of the MDB. It constitutes about 60% of the total lines of code in the command analysis module. The high complexity measurements shown for this procedure agree well with an intuitive understanding of the procedure's size and central role in command analysis.

.MESSG

This procedure is used to print messages to the user and is called from several places in the MDB. The complexity values for this procedure confirm the fact that each of the complexity metrics only measures a different dimension of complexity. As one can see from Table 3, there is no logical ripple effect (Yau metric) associated with this procedure and it has a very small cyclomatic complexity. However, all the other metrics are quite large for this procedure. A quick look at the program logic in MESSG shows that the the procedure has a computed GOTO statement that determines which one of many message is to be printed. This procedure is really not a problem component in the system.

Procedures like MESSG have been observed in other systems that we have analyzed. Such procedures are a warning that a metric-based evaluation of a system should not be done blindly. Experience and design knowledge are still needed to interpret the metric information. The metrics can focus, but cannot replace, the knowledge and experience of designers.

Analysis Of New MDB

As mentioned earlier the students in an advanced graduate course in information systems redesigned the MDB. It was of interest to find out if the metrics could be used in the early phase of a redesign activity. After the metrics were gathered, the procedures were sorted by the complexity values to identify the outliers. Table 5 shows the outlier procedures that had the highest complexity values for most metrics. Since the internal code for each procedure was not complete, one cannot attach much significance to the code metric values. Also, Yau and Collofello's logical stability metric is not a very reliable indicator of complexity when applied to incomplete source code like the New MDB. The reason is that Yau and Collofello's metric considers the ripple caused by changes to the local variables and this ripple effect cannot be identified in a procedure that does not specify all the local variables that will eventually be used when the system is complete.

	code metrics			structure metrics			
procedure	lines code	effort	cyclo. compl.	McClure	info. flow	Wood field	Yau
Valuemgt	27	118	6	*	$3x10^{7}$	348	.5
Process	22	106	8	*	$8x10^{5}$	177	.5
Dmldriver	24	100	6	*	$1x10^{6}$	212	0
AccMgmt	4	32	3	*	$9x10^{5}$	95	0

Table 5: Complexity Outliers (New MDB)

As it can be seen from the table, the procedure VALUEMGT had a very high information flow complexity, as compared to any of the other procedures. This fact was brought to the attention of the project administrator for the New MDB. Interestingly enough, the author was told by the project administrator that he had an intuitive feeling that the procedure was not well structured and could become a bottleneck in the future. It is obvious that the information flow complexity value for that procedure very strongly indicates a possible poorly structured design. One useful result of this evaluation was the decision by the project managers to redesign the procedure since it performs a major function of the New MDB.

In the words of the project director Dr. Rex Harston,

"The metrics generated for the New MDB spotted the modules which were causing the most confusion during the design process, allowing for appropriate adjustments during design. It is, therefore, my feeling that complexity measurement tools ought to be available as part of a system design support environment"

As we indicated at the beginning of this paper, it is our belief that this type of positive subjective experience by knowledgeable system designers and administrators is as valuable as objective statistical experiments. Both types of studies and results are needed to provide the compelling evidence necessary to gain industry acceptance of a measurement approach to controlling the software development and maintenance processes.

A significant observation was made by one of the reviewers about the subjective evaluations reported in this paper. The subjective evaluation does not indicate that the *factors* taken into account by a metric were responsible for the perceived complexity or the difficulties experienced in performing the maintenance tasks. For example, suppose there is a component with a high information flow measure which is subjectively known to be difficult to modify. Can the difficulty of maintenance be related to the large number of connections which this component has with other components? Or is it merely a fortuitous circumstance that the component as high information flow? While the consistency of our experience in this study leads us to believe that we have not been merely lucky in identifying complex components, we cannot, at this point, offer experimental evidence to soundly justify this conviction.

V. Conclusions

An analysis of three different versions of the same single system enabled us to achieve some insight into the potential use of software complexity metrics to assess and control software maintenance activities. Because we have only had the opportunity to apply this approach to a single system great care must be taken in interpreting the results of this analysis. Without loosing sight of this important limitation the encouraging results of this study should, at least, argue for repeated experiments of this form on other systems.

This study has also confirmed the results obtained in previous work with respect to the distinction between the code and structure metrics. This distinction was evident in that the maintenance changes to components might dramatically alter the values of metrics in one class of metrics without changing materially the values of metrics in the other class. This study also confirmed that it is useful to examine carefully those components which are "outliers" of one or more metrics. These outliers appear to be few in number in realistic systems and also seem to be fruitful places in which review and quality assurance resources might well be invested.

One of the useful byproducts from this research was the experience gained in building the software metric analysis tool that computes the set of metrics used in this study. A more flexible and friendly version of this tool is now under development.

References

[1] V. Basili, "Evaluating software development characteristics: Assessment of software measures in the software engineering laboratory", *Proceedings: Sixth Annual Software Engineering Workshop*, NASA Goddard, December 2, 1981.

[2] V. Basili, R. Selby, and T. Phillips, "Metric Analysis and Data Validation Across Fortran Projects", *IEEE Transactions on Software Engineering*, Vol. SE-9, No. 6, pp. 652-663, November 1983.

[3] L. Belady, and M. Lehman, "A Model of Large Program Development", *IBM System Journal*, No. 3, pp. 225-252, 1976.

[4] B. Boehm, "Quantitative Evaluation of Software Quality", *Proceedings: 2nd International Conference on Software Engineering*, San Francisco, CA, pp. 592-605, October 1976.

[5] B.Boehm, "Software Engineering - As It Is", *Proceedings: 4th International Conference on Software Engineering*, Munich, Germany, pp. 11-21, September, 1979.

[6] P. Brown, "Why Does Software Die?", *Life-Cycle Management*, Infotech State of the Art Report, Series 8, No. 7, 1980.

[7] J. Canning, *The Application of Software Metrics to Large-Scale Systems*, Ph.D. Thesis, Department of Computer Science, Virginia Polytechnic Institute, April 1985.

[8] W. Curtis, et.al., "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics", *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 2, pp. 96-104, March 1979.

[9] M. Halstead, *Elements of Software Science*, Elsevier North-Holland, Inc., New York, N.Y., 1977.

[10] S. Henry, *Information Flow Metrics for the Evaluation of Operating Systems' Structure*, Ph.D., Thesis, Iowa State University, 1979

[11] S. Henry, and D. Kafura, "Software Structure Metrics Based on Information Flow", *IEEE Transactions on Software Engineering*, Vol. SE-7, No. 5, pp. 5109-518, September 1981.

[12] S. Henry, D. Kafura, and K. Harris, "On the Relationships Among Three Software Metrics", *Performance Evaluation Review*, Vol. 10, No. 1, pp. 81-88, Spring 1981.

[13] S. Henry, and D. Kafura, "The Evaluation of Software Systems' Structure Using Quantitative Software Metrics", *Software: Practice and Experience*, Vol. 14, No. 6, pp. 561-573, June 1984.

[14] D. Kafura, S. Henry, "Software Quality Metrics Based on Interconnectivity", *The Journal of Systems and Software*, Vol. 2, pp. 121-131, 1981.

[15] D. Kafura, J. Canning, and G. Reddy, "The Independence of Software Metrics Taken at Different Life-Cycle Stages", *Proceedings: 9th Annual Software Engineering Workshop*, NASA Goddard, pp. 213-222, November 1984.

[16] D. Kafura, and J.T. Canning, "A Validation of Software Metrics Using Many Metrics and Two Resources", *Proceedings: 8th International Conference on Software Engineering*, London England, pp. 378-385, August 1985.

[17] M.M. Lehman, "On Understanding Laws, Evolution and Conservation in the Large-Program Life Cycle", *Journal of Systems and Software*, Vol. 1, No. 3, pp. 213-232, 1980.

[18] T.J. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. Se-2, pp. 308-320, December 1976.

[19] C. McClure,"A Model for Program Complexity Analysis", *Proceedings: 3rd International Conference on Software Engineering*, Atlanta GA, pp. 149-157, May 1978.

[20] H. Rombach, "Software Design Metrics for Maintenance", *Proceedings: 9th Annual Software Engineering Workshop*, NASA Goddard, pp. 100-135, November 1984.

[21] H. Rombach, "Impact of Software Structure on Maintenance", *Proceedings: Conference on Software Maintenance*, November 1985.

[22] V. Shen, et.al., "Identifying Error-Prone Software -- An Empirical Approach", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 4, pp. 317-323, April 1985.

[23] S. Woodfield, *Enhanced Effort Estimation by Extending Basic Programming Models to Include Modularity Factors*, Ph.D. Thesis, Purdue University, 1980.

[24] S. Yau, and J. Collofello, "Some Stability Measures for Software Maintenance", *IEEE Transactions on Software Engineering*, Vol., SE-6, No. 5, pp. 545-552, 1980.