To successfully manage information systems in the cost-conscious 1990s and beyond, IS managers must become more effective in delivering their services. They should be able to identify and replicate practices that result in higher productivity, lower costs, and higher quality and correct or eliminate ineffective practices. This process requires a continuous flow of information to determine the quality of the IS activities being managed. Subjective guesses are not accurate enough and are open to dispute. Solid, objective measurements are required.

This chapter discusses how to use software metrics as a basis for making management decisions. It describes the sources and methods of collection for the data presented and demonstrates how cost and frequency-of-change information can be used with technical metrics to provide comparisons of cost for quality and productivity. The chapter also uses historical data collected from over 12,000 programs in more than 65 companies to illustrate the current state of IS software and conversely to identify the desirable characteristics in a program or system. Programming standards are recommended, based on the metrics described, and steps for implementing a software measurement program are outlined.

WHY MEASURE?

Most IS departments do not have formal methods for measuring their programming and maintenance activities. As a result, the management of these departments would be hard pressed to answer questions such as:

- Has quality or productivity improved or declined in the last five years? By how much?
- Which programs or systems require a disproportionate amount of programming resources to maintain?
- What are the desirable characteristics in a program or system?
• Does a new software tool or development technique actually improve quality or productivity?
• Does increased quality actually lead to decreased cost?

All these questions, and more, could be answered through the implementation of a formal measurement program.

A comprehensive measurement program would combine software quality metrics, frequency, and quantity of change information with cost or effort information to provide an accurate picture of current programming practices. This information would be used to:

• Define Quality. Which practices are effective? What are their characteristics? The answers to these questions provide a real-world definition of quality. For example, small programs cost far less to maintain than large programs. Therefore, small is beneficial in programming.
• Quantify Quality. Once a definition of quality has been identified, the IS environment can be measured to determine the volume of characteristics. For instance, how many of the programs in a general ledger application are too large to maintain effectively?
• Set a Baseline. Each compilation of measurement information provides a baseline of comparison for future runs. This baseline can be used for identifying trends, setting improvement objectives, and for measuring progress against those objectives. Continuing the example, after one year of improvement activities, what percentage of the general ledger programs are still too large?

Combining cost and effort data with software quality metrics is particularly valuable for management decision making. It enables IS managers to determine relative productivity rates and to make cost-quality comparisons. Productivity rate information helps identify productivity bottlenecks and provides a foundation for accurate task estimation. Cost and quality information provides a basis for cost justifying quality improvement activities.

Maintaining a historical data base of key measurement data enables long-term validation of the decisions made from the measurement data. Quality and productivity trends can be determined, and the use and collection of the metrics information can be fine tuned if necessary. The benefits can be periodically reanalyzed to ensure that they are matching their cost-justification projections. Finally, as the base of historic data grows, it becomes more valuable for verifying task estimates.

METHODS USED IN THIS STUDY
There are three major categories of data used in this chapter: technical quality data, cost and effort data, and frequency-of-change data. This data
is gathered through a variety of automated and manual methods. Much of the data presented here was gathered from Keane, Inc.'s reengineering consulting practice. Keane's analysis combines its ADW/INSPECTOR metrics with other statistics, such as programmer cost, maintenance effort by task, frequency of change, and failure rates, to assist IS management in making strategic decisions about its application portfolio.

Technical Quality

The software metrics data presented in this chapter is a combination of statistics that have been gathered over five years by both Language Technology, Inc. (now a part of KnowledgeWare) and Keane, Inc. The software quality statistics are derived primarily from a data base of more than 150 software metrics collected using an automated COBOL analysis tool. This data base was originally developed by Language Technology from data contributed by more than 65 customers. The data base encompasses well beyond 12,000 COBOL programs consisting of 23 million lines of code.

Cost and Effort

Cost and effort data are essentially equivalent. For example, effort can be converted to cost by multiplying the number of hours expended by the average cost rate for the programmers. Gathering this data varies widely from company to company. Common sources are time-accounting systems, budgets, or task-tracking systems. Ideally, programmer time is charged to specific projects and is broken down by type of task. This is rarely available, so in most cases, specific cost allocations have to be extrapolated from available information.

Frequency of Change

Determining productivity rates and isolating the most volatile, and hence high-payback, areas of an application system require the availability of frequency- and volume-of-change information. This type of information can take many forms, depending on the company’s practices. For example, if the company measures function points, volume of change is the number of function points added, deleted, or modified over a period of time. Another method is to track user change requests. These methods are less accurate from a programmer’s point of view, but they have the advantage of having more meaning to the business areas. More commonly, however, this data is derived from the company’s library management system. If set up properly, the library management system can provide reports on the number of changes made for each module contained within it, and often it can state the size, in lines of code, for each of those changes.
METRICS AND COST

Many companies are content with using only technical metrics to evaluate their programming and maintenance practices. Although this gives them the ability to evaluate the quality of their practices, it does not allow the correlation of quality to cost. This correlation provides a wide range of information that is invaluable to IS management. Some key examples of analyses that are accomplished in conjunction with cost are:

- Productivity measurement.
- Comparison of applications by effectiveness.
- Establishing the value of quality.
- Isolation of areas of high cost and high payback.
- Task estimate validation.
- Cost justification.

Examples of these analyses are illustrated in the following sections.

Productivity Measurement

Productivity measurement is a sensitive topic among those who are to be measured. Given the level of accuracy of the numbers available in most companies, productivity comparisons are best done at the application level rather than at the individual level. The basis for measuring productivity is effort per unit of work. Using cost in place of effort is equally valid. Potential units of measure include function points, statements, lines of code, change requests, or any other unit of work routinely measured by the company. Each has its own strengths and weaknesses, but any unit works for gross measures. The best unit that is easily available should be used because 80% of the value can be derived with the first 20% of the effort. A typical unit used by Keane is the number of lines of code changed per year. Changed, in this case, is defined as added, modified, or deleted. This number has the advantage of being relatively easy to obtain through ADW/INSPECTOR and a library management tool.

Taking the overall cost or effort expended on maintenance for a given application and dividing it by the volume of changes leads to a productivity factor. For example, application A may have a productivity factor of $10 per line of code changed. Improvements in productivity are reflected as a corresponding drop in the per-unit cost.

Comparison of Applications by Effectiveness

As Exhibit 1 illustrates, the productivity factor may be used to compare applications against each other. Five applications that were measured during a portfolio analysis are ranked against each other by their relative productivity factors. This relative ranking provides a challenge to management to determine why productivity is better for one application than
another. Is it due to technical quality, personnel, or maintenance records? Answering these questions has inestimable value. If the factors causing low costs on ACP could be duplicated on CSA, there would be an almost fourfold increase in productivity.

**Establishing the Value of Quality**

*Exhibit 2* is a combination of the productivity statistics in *Exhibit 1* and average application quality statistics. The quality is measured by the average ADW/INSPECTOR composite score for all the programs within the application. This metric will be explained in more detail later in this chapter; in short, the score ranges from 0 to 100, with 100 being ideal.

The combination of cost and quality data as illustrated in *Exhibit 2* demonstrates the strong correlation between application quality and the cost of maintenance. Even small increases in average quality generally result in measurable reductions in the cost of maintenance. This correlation between application quality and the cost of maintenance is typical when multiple applications within a company are measured. Performing this type of correlation is valuable for developing actual dollar benefit estimates for quality improvement efforts.

**Isolation of Areas of High Cost and High Payback**

Some programs within an application consume a much higher proportion of maintenance resources than other programs. These programs may be isolated by distributing the cost and change data down to the individual program level. Whereas programmers can generally identify the top one or
two programs requiring the most maintenance effort, the overall distribution of maintenance cost and effort is often surprising. Interestingly, between 3 and 5% of the programs in a given application account for 50 to 80% of the overall maintenance effort. Those numbers are remarkably consistent across many companies. Exhibit 3 illustrates this cost distribution for the five applications used in the previous examples. Twenty programs, or approximately 3% of the total number of programs, account for 50% of the overall cost of maintenance.

To develop this graph, yearly maintenance costs were allocated to each program within the applications. The programs were sorted by cost, and the graph line charts the accumulation of these costs (i.e., from the left, point 1 is the percentage cost of the most expensive program; point 2 is the sum of the two most expensive programs). Knowing where effort is actually expended allows improvement activities to be targeted to those programs in which the value would be the highest. Of the programs in this example, 28% had no maintenance effort expended on them at all. Consequently, they would have no payback if improved.

**Task Estimate Validation**

Keeping cost statistics and in particular developing a productivity factor is valuable for use in estimating the effort and cost of maintenance changes. Once the approximate size of a maintenance change is known, it can be multiplied against the productivity factor to provide an approximate scope of
effort. This may be used directly or as a validation for other estimation methodologies. If cost or effort data is correlated against other factors, such as size or technical quality, these factors can be used as modifiers when estimating. For example, Exhibit 4 illustrates the cost per statement changed based on the number of statements changed in a particular modification for a client.
As would be expected, maintenance modifications that encompass only a few lines of code are more expensive per statement changed than larger modifications. This reflects the overhead for finding, inserting, testing, and placing the modification into production. The overhead factors are generally uniform relative to the size of the modification. Knowing this curve allows the calculation of an adjustment factor that can be applied to estimates for small changes.

Cost Justification

Using methods similar to the estimation technique described above, cost and productivity factors can be used for developing cost/benefit analyses for justifying new productivity tools or methodologies. For instance, the comparison of the five applications in Exhibit 2 can easily be used to justify quality improvement efforts. The only additional piece of necessary information is the total volume of changes. In the case of application CTN, this volume is about 114,000 statements per year. This application costs $9.09 per statement maintained, whereas application ACP costs $4.37. If the quality of CTN could be improved to meet or exceed that of ACP, one would expect that its cost of maintenance would be similar to ACP. Thus, if the improvement activities resulted in a per-statement drop in cost of $4.72 ($9.09 – $4.37), the anticipated benefit would be about $538,000 per year (114,000 $4.72), Comparing this benefit to the cost of making the necessary improvements would provide the cost justification.

METRICS AND QUALITY

Cost metrics describe the value of a particular activity, whereas technical metrics evaluate the quality of its implementation. This industry has over 30 years of experience in the effectiveness of programming practices, yet practitioners still argue about the merits of such advances as GO TO-less programming. The metrics in the Language Technology data base confirm most theories on effective programming. This data should be used to define the standards by which programming is performed. Even though the data is based on COBOL programs and programmers, the insights are not limited to that environment. Human abilities for handling complexity, for example, are not language, or even programming, dependent.

Background

Some important definitions are necessary before embarking on a discussion of the metrics and their meanings. Perhaps the most important definition is that of the ADW/INSPECTOR composite score. As its name implies, this metric is a composite of a number of other measures. A 0-to-100 scale is used to provide a quick ranking of programs by quality. This metric has a consistently high correlation with other measures of quality, and it is an
accurate predictor of relative maintenance effort, and therefore cost, between programs. The composite score is calculated as follows:

- Degree of structure (based on McCabe’s essential complexity): 25%.
- Degree of complexity (based on McCabe’s cyclomatic complexity): 25%.
- Number of ALTER verbs: 10%.
- Number of GO TO (i.e., non-exit) verbs: 10%.
- Number of fall throughs (i.e., to non-exit statements): 10%.
- Number of active procedure exits: 10%.
- Number of recursive procedures: 10%.

A perfect program receives 100 points (totally structured, not complex, and no ALTERs or GO TOs).

Other important definitions are:

- **Active Procedure Exits.** These result when a PERFORMed procedure within a COBOL program is exited by a GO TO while still active from a PERFORM. Unpredictable program behavior can result when these procedures are reached via another control path in the program.
- **Recursive Procedures.** These are the result of a PERFORMed procedure being re-PERFORMed before completing a previous PERFORM. It too can cause unpredictable behavior.
- **McCabe Metrics.** These will be discussed in the following sections.
- **Lines of Code.** These are the specific number of lines of 80 column COBOL text in a program. Statements are the count of COBOL procedural verbs, such as MOVEs or ADDs.

**Industry Averages**

A major advantage of having a large multicompany data base of metrics is the ability to compare a specific company’s application software with the rest of the industry. This provides a relative measure for that software against its “average” peers.

**Exhibit 5** contains the characteristics for the average COBOL program. The characteristics of the average program may surprise many programmers: It is small in size and its quality is high. These effects are the result of mixing older COBOL programs with more newly developed code. As with averages in general, this obscures the very different characteristics of these two distinct categories.

This distinction between new and old code is illuminated when the programs within the data base are distributed by composite score. This distribution is shown in **Exhibit 6**.
The score distribution in Exhibit 6 is bimodal (i.e., there are two separate peaks). The peak at the far left is that of the average old application; its average program is in the 35-point range for the composite score. The peak at the far right mostly represents newer programs. Many new programs are created in conformance with rigorous structured programming standards and tend to be high in quality.

Over time, however, this quality tends to degrade through the effects of multiple coding changes unless specific quality efforts are in place. This decline is made apparent by the number of programs in the 81-to-90-point range.

The differentiation between the newer code and the older code becomes even more clear when the data is categorized by structure and complexity.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Industry Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite Score</td>
<td>60</td>
</tr>
<tr>
<td>Unstructured</td>
<td>43%</td>
</tr>
<tr>
<td>Complex</td>
<td>47%</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>1902</td>
</tr>
<tr>
<td>Statements</td>
<td>666</td>
</tr>
<tr>
<td>GO TOs</td>
<td>52</td>
</tr>
<tr>
<td>Fall Throughs</td>
<td>23</td>
</tr>
<tr>
<td>Recursion</td>
<td>3</td>
</tr>
<tr>
<td>Active Exits</td>
<td>12</td>
</tr>
</tbody>
</table>

Exhibit 5. Characteristics for the average COBOL program.

Exhibit 6. Characteristics for the average COBOL program.

The score distribution in Exhibit 6 is bimodal (i.e., there are two separate peaks). The peak at the far left is that of the average old application; its average program is in the 35-point range for the composite score. The peak at the far right mostly represents newer programs. Many new programs are created in conformance with rigorous structured programming standards and tend to be high in quality.

Over time, however, this quality tends to degrade through the effects of multiple coding changes unless specific quality efforts are in place. This decline is made apparent by the number of programs in the 81-to-90-point range.

The differentiation between the newer code and the older code becomes even more clear when the data is categorized by structure and complexity.
Exhibit 7 divides programs into four quadrants using unstructured percentages and complex percentages. The newer programs tend to be concentrated in the lower left quadrant. They are structured (i.e., under 25% unstructured), and not complex (under 25% complex). Whereas 30% of the programs fall into this desirable category, they make up only 11% of the total number of COBOL statements, the measure of actual code content.

Conversely, the older programs tend to be concentrated in the upper right quadrant and are both unstructured and complex. The upper right quadrant contains the worst 50% of the programs. Using statements as the measure, those programs make up 74% of the physical volume of COBOL code.

Even more instructive is selecting the average program from each quadrant. These averages are illustrated in Exhibit 8. In this exhibit, overlaps are the same as active procedure exits, and procedure size is the number of lines of code in the COBOL program’s procedure division.

**Structured/noncomplex.** As mentioned earlier, these are the desirable programs. They have a very high average composite score of 93 points and a very low unstructured percentage and complex percentage. This is particularly interesting given the 25% cutoff for unstructured percentage and complex percentage; the low numbers are well below the expected average of about 12.5% for an even distribution. Size is an important factor in the quality of these programs. Desirability in programs is achieved through the combination of small size, high structure, and no GO TOs. Despite the single GO TO in the average, the majority of the programs in this category have no GO TOs at all.

**Unstructured/noncomplex.** These are the rarest programs in the data base sample due to the difficulty of creating an unstructured program without

<table>
<thead>
<tr>
<th></th>
<th>Unstructured</th>
<th>Structured</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5% of Total</td>
<td>1% of Total</td>
</tr>
<tr>
<td>Programs</td>
<td>678</td>
<td>6,285</td>
</tr>
<tr>
<td>Lines</td>
<td>347,083</td>
<td>16,387,023</td>
</tr>
<tr>
<td>Statements</td>
<td>89,864</td>
<td>6,131,621</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Noncomplex</th>
<th>Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11% Total</td>
<td>14% Total</td>
</tr>
<tr>
<td>Programs</td>
<td>906,110</td>
<td>1,200,590</td>
</tr>
<tr>
<td>Lines</td>
<td>3,070,197</td>
<td>3,501,542</td>
</tr>
<tr>
<td>Statements</td>
<td>3,769</td>
<td>1,772</td>
</tr>
</tbody>
</table>
measurably increasing its complexity. This is generally possible in only very small programs, which is demonstrated by the fact that these programs make up 5% of the total in number but only 1% of the physical volume in statements.

These programs are surprisingly unstructured given their very small size. This lack of structure is caused by the increase in the average number of GO TOs and fall throughs. The drop in quality reflected in this quadrant results in an increase in the number of lines of dead code. Dead or inexecutable code is generally introduced accidentally when a poorly understood program is modified. These programs could easily be improved by simply running them through a structuring tool.

**Structured/complex.** This quadrant contains programs that are highly complex despite being structured. Some of these programs are necessarily complex due to the nature of the tasks they are performing; others are needlessly complex because of poor design. In either case, studies have shown that both error rates and testing effort increase as complexity increases. Complexity is highly correlated to program size. This is demonstrated by the significantly larger size of these programs as compared to the programs in the noncomplex categories. The presence of additional numbers of GO TOs and fall throughs also results in an increased complexity. The best method for reducing complexity in a structured program is to subdivide complex portions of that program into smaller noncomplex portions whenever

Exhibit 8. Average programs per structure and complexity.

<table>
<thead>
<tr>
<th>Metric</th>
<th>Structured Noncomplex</th>
<th>Unstructured Noncomplex</th>
<th>Structured Complex</th>
<th>Unstructured Complex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Composite Score:</td>
<td>93</td>
<td>63</td>
<td>78</td>
<td>34</td>
</tr>
<tr>
<td>Unstructured %:</td>
<td>1</td>
<td>66</td>
<td>4</td>
<td>77</td>
</tr>
<tr>
<td>Complex %:</td>
<td>3</td>
<td>1</td>
<td>50</td>
<td>78</td>
</tr>
<tr>
<td>Size in Location:</td>
<td>815</td>
<td>512</td>
<td>1976</td>
<td>2684</td>
</tr>
<tr>
<td>Statements:</td>
<td>240</td>
<td>133</td>
<td>678</td>
<td>976</td>
</tr>
<tr>
<td>Procedure Size:</td>
<td>451</td>
<td>236</td>
<td>1227</td>
<td>1565</td>
</tr>
<tr>
<td>Go Tors:</td>
<td>1</td>
<td>9</td>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>Fall Throughs:</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>43</td>
</tr>
<tr>
<td>Recursion:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>OverLaps:</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>Deadcode:</td>
<td>6</td>
<td>10</td>
<td>9</td>
<td>36</td>
</tr>
</tbody>
</table>
possible. Breaking larger programs into multiple smaller modules also reduces complexity.

Unstructured/complex. The programs in this quadrant are the classic old COBOL programs. The composite score drops dramatically as the result of increased complexity and decreased structure. Size has increased significantly, making these programs even more difficult to understand. Particularly alarming is the massive increase in poor coding practices, such as GO TOs, fall throughs, and recursion. These are averages, however; there are programs that are considerably worse in this category. This major drop in quality results in programs that are very hard to maintain and have significantly higher failure rates. This is reflected in the increased number of lines of dead code. Programs in this category are candidates for reengineering improvement activities to lower the cost and effort of maintenance.

Program Size Factors

Perhaps the most critical determinant of a program’s quality and maintainability is its size. It is theoretically possible to produce a large, well-structured, noncomplex program; however, analysis of the statistics in the metrics database show that this is an extremely rare occurrence. In fact, size in statements has a strong correlation with every negative programming practice. Size is not a component of the composite score, but as Exhibit 9 illustrates, the composite score increases as the size decreases.

In fact, programs in the ideal composite score range of 93 to 100 points average only 240 statements in size. This size appears to be a boundary in program understanding. It appears consistently in the analysis results from
company to company. In one client engagement, multiple years of ADW/INSPECTOR metrics were available, allowing the comparison of the effects of maintenance changes over time. When compared by composite score, programs under 240 statements in size either retained their score through multiple maintenance changes or actually improved in score. Conversely, programs over 240 statements generally declined in composite score. This effect appears to be caused by the relationship of complexity and size. Exhibit 10 is a graph of size, structure, and complexity correlation.

Again, although structure and complexity should be independent of size, they are highly related in practice. The sheer size of large programs makes them difficult to understand fully and often forces programmers to make localized program modifications that actually increase complexity and degrade structure. This tendency accelerates over time until the large program becomes unmaintainable.

As would be expected, size is related to the cost of maintenance. The graph in Exhibit 11 is from another Keane portfolio analysis engagement. The bars represent the number of programs in each size category. The graph line shows the total maintenance expenditure in dollars for all the programs in each respective column.

The highest expenditure of dollars is on the smallest number of programs, those with over 1,000 statements in size. The second-highest total expenditure is for the programs in the under-250-statement category; however, as the bar indicates, they make up a disproportionate number of the
programs. This data is more dramatic when the average maintenance cost per program is graphed in Exhibit 12.

McCabe Metrics

The metrics of T. J. McCabe are widely used to measure the complexity and structure of programs. Numerous studies have shown that these metrics are
accurate predictors of program defect rates and program understandability. There are two separate metrics: cyclomatic complexity and essential complexity. Both metrics are based on measuring single-entry, single-exit blocks of code. In a structured COBOL program, these blocks of code are individual paragraphs. In a convoluted, unstructured program, the entire program may be one single-entry, single-exit block.

**Cyclomatic complexity.** This is the measure of the number of test paths within a given single-entry, single-exit block of code. The number of defects in a block of code greatly increases when the number of test paths in that block exceeds 9. This is based on the theory that the average person can assimilate 7 plus or minus 2 (i.e., 7 +/-2) pieces of detail at a time in short-term memory. In COBOL programs, the number of test paths can be estimated by counting the number of IF statements in the block of code being measured. Adding 1 to this number gives the total number of unique test paths in that block of code. Therefore, as long as there are no more than 9 IF statements in each single-entry, single-exit block of code, it will meet the McCabe standards.

The 7 +/-2 levels of detail principle represented by cyclomatic complexity comes through in Exhibit 13. This graph from the metrics data base shows the distribution of the number of statements per paragraph across the entire data base.

The standard deviation in this graph is 8 +/-4, showing that programmers naturally tend to limit themselves to the McCabe cyclomatic constraints.

**Essential Complexity**

Essential complexity measures the degree of structure in a block of code. Essential complexity is measured by reducing all structured constructs (i.e.,
conditions, sequence, and loops) out of a given block of code, then measuring the remaining complexity. If no complexity remains, the piece of code is structured. Any remaining complexity is unessential (i.e., it could be removed by structuring that piece of code). Unstructured percentage is the sum of the number of unstructured paths divided by the total number of paths in the program. Essential complexity is used to measure if a particular program is in compliance with structured standards. It also predicts the ease with which a program can be modularized. Exhibit 14 illustrates this principle in an unstructured program.

The program in the real life portion of the illustration is primarily unstructured. The first two and last two paragraphs are single-entry, single-exit, but the entire middle section of the program is so interwoven with control flow that it is effectively one large block. The structure chart represents a view of the program if it were imported into a CASE tool. The middle block in the structure chart, shown as paragraph Initial-Read, actually contains all the code from the convoluted middle section, as shown in Behind the Scenes. If this program were to be modularized for reusability, this large block of code could not be easily subdivided unless it was structured first. Ideal programs should be totally structured as measured by McCabe’s essential complexity.

**GO TOS AND GO TO EXITS**

Despite the advent and supposed acceptance of structured programming, GO TOS are still used. ADW/INSPECTOR can differentiate between GO TOS and GO TO EXITs, which allows them to be examined separately. GO TO EXITs can theoretically be structured, so they are not penalized in the composite score. When GO TO non-exits are used within a program, they greatly increase its complexity. This is shown in Exhibit 15.

This complexity increases disproportionately as the number of GO TOS increases. Exhibit 16 shows the effect of GO TOS on the Composite Score.

The number of GO TO nonexits accounts for 10 points of the composite score. As can be seen from this graph, their effect on program quality is far beyond those 10 points.

Many programmers argue in favor of the use of GO TO EXITs. This construct was originally used to reduce nesting of IF statements within paragraphs. This is no longer necessary in structured programming. Structuring tools can be used to automatically control nesting levels, and the use of structured CASE constructs eliminates much of the complexity involved in the nesting of related conditions. If implemented correctly, GO TO EXITs should have no effect on structure and complexity. In practice, this is not true. Exhibit 17 shows the number of GO TO EXITs graphed against complexity.
Exhibit 14. The view within the CASE tool.
Although the correlation is not as strong with GO TO non-exits, GO TO EXITs increase program complexity. This appears to be the result of two factors. First, GO TO EXITs are legitimate as long as they go to the correct exit. Unfortunately, their very existence invites accidental misuse. Second, the presence of any GO TOs tends to beget other GO TOs. For these reasons, it makes the most sense to avoid the use of any GO TOs when programming.
ALTERS

The ALTER verb is rarely seen in COBOL nowadays. A vestige from the assembly programming days, it appears only in some of the oldest programs. When ALTERs are present, their negative effect on quality amply justifies their removal. This may be done manually or automatically with a structuring tool. As with GO TOs, ALTERs comprise 10 points of the composite score. Again, however, their effect extends far beyond those 10 points. This is shown in Exhibit 18.

The presence of any ALTERs within a program tends to push it below 30 points in the composite score. ALTERs also greatly increase the number of test paths within the program. The worst example in the data base was a
1,500-line program that contained 508 ALTER statements. It had 4,718 separate test paths as measured by McCabe’s cyclomatic complexity.

**METRICS AND STANDARDS**

With more than 30 years of industry experience in COBOL programming, one would expect that the characteristics that make one program easier to maintain than another would be well known and incorporated into IS practices. Unfortunately, that has not been the case, in part because of the lack of agreement about what makes a desirable program. The data presented in this chapter should shed some light on some of the most important characteristics.

The maintainability of COBOL programs can be greatly enhanced, thereby lowering the cost and effort of maintenance, by following a few simple standards. The key to excellence in COBOL programs is understandability. If a program can be easily understood, it can be modified quickly, can be tested more thoroughly, and will be less likely to contain defects.

When all the data is analyzed, the three crucial characteristics that lead to understandability are size, structure, and modularity.

- **Size.** Once a program exceeds a certain size, it becomes difficult to understand and maintain just due to sheer mass. Changes are made to large programs without understanding the context of the whole program. This introduces poor coding practices and increases complexity. Large programs tend to contain many functions. Modifications to any of these functions require analysis and testing of the entire program, further increasing maintenance effort. Thus, small is better in programming. As the data in this chapter indicates, the ideal size limit appears to be 240 or fewer COBOL statements.

- **Structure.** Programs consisting of well-organized, single-function paragraphs are the easiest to understand and maintain. Strictly following the dictates of structured programming helps ensure that programs meet these standards. Further, if a program is structured, by definition it does not contain any poor coding constructs, such as GO TOs, fall throughs, or recursion.

- **Modularity.** Each paragraph in the program should consist of a single business function, and it should have only one entry and one exit. Further, the business functions must be modularized such that each paragraph consists of 8±4 COBOL statements. If the program is structured, this rule ensures that the program is not complex by the McCabe standard. This simplifies testing, enables the location and reuse of business functions, and enhances the value of importing the program into a CASE environment.
Each of the three characteristics described above should be incorporated into IS coding standards for both new development and maintenance. Adherence to these standards can be easily measured with an analysis tool like ADW/INSPECTOR using the metrics discussed in this chapter. Exhibit 19 contains the optimum metric values for measuring compliance with these recommended COBOL programming standards.

For new development, these standards can be directly applied as acceptance criteria for newly written programs. Following these standards ensures that these new programs are easy to modify in the future.

For maintenance and enhancement projects, these standards become targets. At the very minimum, programs should be measured after each modification to ensure that quality is maintained. Ideally, programmers can attempt to slowly improve programs to get closer to these standards each time they make a maintenance change.

Finally, these standards are a goal for reengineering or other improvement activities to existing code. These efforts should be targeted at only the most highly maintained 3 to 5% of the programs to ensure payback. Attempting to reengineer all code to this standard would be cost prohibitive, and it would not provide any benefits on rarely maintained programs.

CONCLUSION
A software measurement program can provide IS managers with valuable insights on how to best manage their scarce resources. As the examples in this chapter demonstrate, metrics can be used to identify specific standards

<table>
<thead>
<tr>
<th>Metric</th>
<th>Recommended</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard Composite Score</td>
<td>93–100</td>
</tr>
<tr>
<td>Unstructured</td>
<td>0%</td>
</tr>
<tr>
<td>Complex</td>
<td>Less than 25%</td>
</tr>
<tr>
<td>Lines of Code</td>
<td>750 or Less</td>
</tr>
<tr>
<td>Statements</td>
<td>240 or Less</td>
</tr>
<tr>
<td>Statements per paragraph*</td>
<td>8–12</td>
</tr>
<tr>
<td>GO TOs</td>
<td>0</td>
</tr>
<tr>
<td>Fall Throughs</td>
<td>0</td>
</tr>
<tr>
<td>Recursion</td>
<td>0</td>
</tr>
<tr>
<td>Active Exits</td>
<td>0</td>
</tr>
</tbody>
</table>

*Average standards per paragraph is calculated by dividing the total number of procedure statements in the program by the total number of paragraphs.

Exhibit 19. Optimum metric values for measuring COBOL compliance
and methods that save money and resources. They can pinpoint when to apply most effectively that knowledge. They can be used to estimate the effort for programming tasks and to quantify the benefits of improvement tasks.

Gaining these benefits requires implementing a software measurement program. Some basic steps are:

**Defining objectives.** Define why a measurement program is needed and what the specific types of questions to be answered are. This identifies what type of data is needed and the scope of the effort.

**Identifying data sources.** A check should be made of the existing data. Is the data complete as required? What data can be collected automatically? What data must be collected manually? Some examples of each type of data should be collected to check its ease of collection, accuracy, and completeness.

**Obtaining tools.** If the company does not already have an analysis tool, it is time to get one. Some tools, such as project tracking and library management software, may be in house but may require specific option settings to collect the necessary data.

**Defining reports.** Copies of the reports and graphs that will be used as output of the measurement program should be mocked up. Distribution frequency should be decided on. Displaying these examples is a key to getting buy-in on the project.

**Pilot testing.** Results from the preceding steps should be tested on a pilot set of applications. It is best to use two or three diverse types of applications to ensure that any potential problems are caught. Training requirements should be identified for those involved in the project.

**Tuning the results.** Any of the collection methods, metrics, and reports should be fine tuned, using the results of the pilot test. This information should be used to estimate effort for the final roll out.

**Developing a roll-out plan.** A plan should be developed to roll out the measurement program across the organization, making sure to include sufficient training.

**Implement.** The process is put in place. Data should be reexamined periodically and tuned accordingly as new information is received. Results should be saved in a data base to allow for comparisons over time.