Estimating the Costs of Software Maintenance Tasks

Harry M. Sneed (MPA)
SES Software Engineering Service
Rosenheimer Landstrasse 37
D-85521 Ottobrunn/Munich, Germany
Tel.: +49/(0)89 609 31 93
Fax: +49/(0)89 608 32 59

ABSTRACT

In order to control the maintenance process, it is necessary to be able to accurately predict maintenance costs. The following paper proposes ways of extending current cost estimation methods to the estimation of maintenance costs, by means of impact analysis, proportional sizing, quality assessment, and productivity adjustment. To illustrate the approach, a practical example from the authors own experience in financial applications is cited.

1. Estimating the Costs of Software Maintenance Tasks

The decision whether to change or enhance a software system is determined by the costs and benefits of that particular maintenance task. There are, of course, situations where the maintainers of an existing system have no choice but to change the system, no matter what it costs. Such is the case when new laws require changes to current business rules or when the operating environment is altered. Even here it is useful to know beforehand what the changes will cost. However, in the case of nice to have new features, a cost estimation is essential.

Take the case when a user requests a different type of report other than what he has received up until now. The first question is whether this new report will help him perform a better job, i.e. to serve his customers better, and, if so, to what extend. What will be the added value of the new report relative to the old one? Will it justify the maintenance effort?

Another case it that of a more flexible user interface. If a user has been working with fixed panels before and now would like to have a mouse-driven graphical interface that allows her to browse various views on her data, how much better or faster will she be able to process transactions. If she can increase her transaction processing rate by more than 50%, then the added value is sure to cover the costs of the enhancement, however, if the productivity increase is only 10%, then the added value is marginal and may not cover the enhancement costs.

In either case two factors are essential to making a decision - one is the benefit and the other the cost of making a change to an existing system. The expected benefit is something the user has to come up with. At best, it should be a monetary value, at worst a relative rating to other desired changes. The estimated cost of the change must come from the maintenance estimation method and it must come quickly and cheaply. There is no time for making a long analysis as might be the case with a new development. In software maintenance, the cost estimate of a change should be made within a day. This means that certain data must be kept available, that there is a predefined process for estimating maintenance costs and that this process is supported by automated tools. The purpose of this paper is to describe what data is needed, what process may be used and what tools are required. At the end it will provide a sample to demonstrate the data, the process, and the tools used. However, before presenting the author's solution, it is first appropriate to review previous approaches to this problem which have been published in the software maintenance literature.

2. PREVIOUS APPROACHES TO ESTIMATING SOFTWARE MAINTENANCE COSTS

One of the first attempts to estimate the costs of software maintenance is described by Barry Boehm in his book Software Engineering Economics.[BOEH81] Boehm maintained that annual maintenance costs can be derived from the initial development effort - MMd - and the annual change traffic - ACT, adjusted by a multiplication factor for the system type - MF.

The basic equation for computing yearly maintenance costs is:

$$\text{Maint-Effort} = MF(MMd * ACT)$$

In the intermediate COCOMO method this effort is further adjusted by a multiplication factor representing the system maintainability. If it is higher than average the maintenance effort will be lower and vice versa.[BOEH83]

Boehm came to the conclusion in a study on maintenance costs in the U.S. Defense Department that maintenance costs are three to five times that of the original development costs, meaning all costs after the initial release of the systems.[BOEH88] His statistics have been quoted by many others in trying to predict life cycle costs. Boehm's approach is, of course, useful at the macro planning level, however, it is of no use in trying to estimate the costs of individual maintenance tasks. Here, a micro planning approach is necessary.

Such an approach was presented by Card, Cotnoir and Goorevich at the 1987 Software Maintenance Conference.[CARD87] They suggested counting the number of lines of code affected by the maintenance action, then relating this count to effort, whereby they are careful to point out that productivity is much lower when making fixes and changes than when writing the code. For their model they used data acquired from maintenance projects at the Network Control Center of the Goddard Space Flight Center. The relationship between lines of code and effort was expressed in the equation

$$SID = 42 + 4.8\ NU + 46.6\ SPRF^{**}$$

where

SID	=	person-days
NU	=	new code units to be implemented
SPRF	=	fixes and changes to be implemented.

42 is the number of days they claim is necessary to test any new release. The effort required to make fixes seems to be unproportionately high, but it is based on real data and has to be accepted. It is doubtful, however, if business data processing shops would be willing to pay so much for maintenance.

Another study which was directed toward assessing maintenance costs was that of Vessey and Weber in Australia.[VESS83] These authors studied 447 commercial COBOL programs to determine to what degree program complexity, programming style, programmer productivity, and the number of releases effects maintenance costs. Surprisingly, they came to the conclusion that program complexity has only a limited impact on repair costs. They also discovered that programming style is only significant in the case of larger programs. The number of releases only affected the costs of adaptive maintenance but not repair maintenance. The more a program is changed, the more difficult it is to adapt. This reinforced the conclusion that increasing complexity drives up adaptive maintenance costs. Thus, it is important to assess program complexity whereas programming style e.g. structuredness and modularity, is not so relevant as many would believe.

Dieter Rombach also studied factors influencing maintenance costs at the University of Kaiserslautern in Germany.[ROMB87] He concluded that the average effort in staff hours per maintenance task is best explained or predicted by those combined complexity metrics which measure external complexity by information flow and which measure internal complexity by length or number of structural units, i.e. nodes and edges. This means that the complexity of the target software should definitely be considered when estimating the costs of maintenance tasks.

More recent studies have been made by Lanning and Khoshgoftaar in relating source code complexity to maintenance difficulty [LANN94] and by Coleman, Ash, Lowther, and Oman to establish metrics for evaluating software maintainability [COLE94]. The latter compute a maintainability coefficient to rate programs as either highly maintainable, i.e above 0.85, moderately maintainable, i.e. above 0.65, or difficult to maintain, i.e. those under 0.65. Both studies come to the conclusion that complexity and maintainability are related.

In recent years, the first studies have been published on the maintainability of object-oriented software. Chidamber and

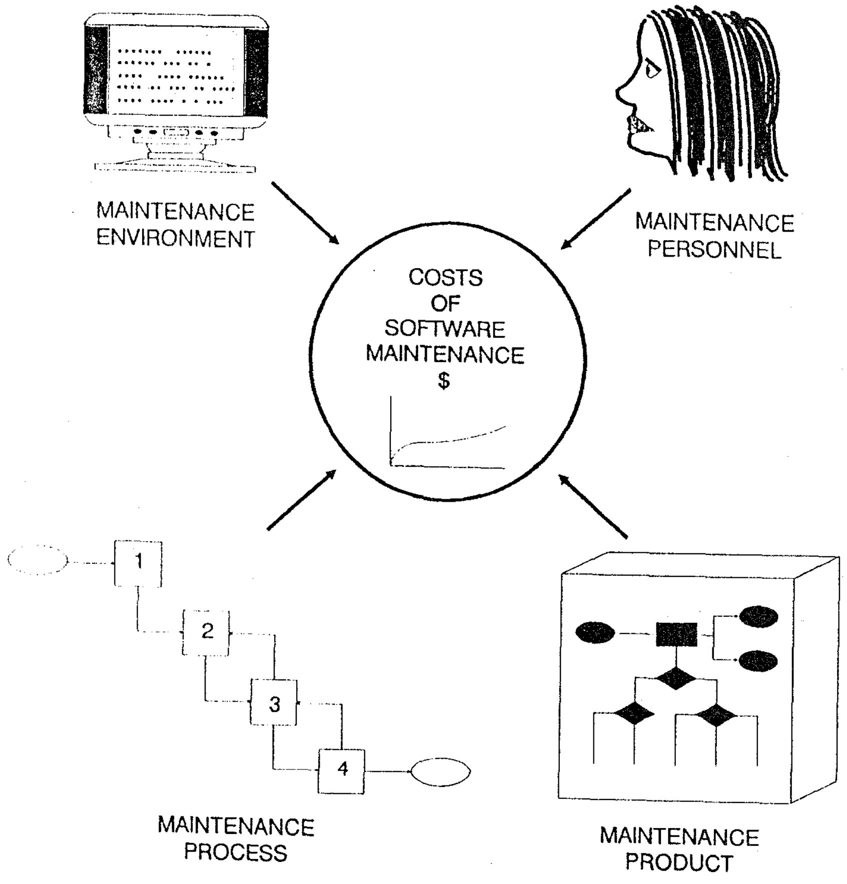

Figure 1
Cost Drivers in
Software Maintenance

Kemerer have developed a set of six metrics for sizing object-oriented software.

- weighted number of methods per class
- depth of inheritance tree
- number of subordinate classes per base class
- coupling between object classes, i.e. the number of messages
- number of possible responses per message
- cohesion between methods in a class, i.e. the number of commonly used data attributes.[KEME94]

These metrics are very important in finding a way to size object-oriented software. Equally important are metrics for determining the complexity of object-oriented software. Wilde and Huitt have identified those features of object-oriented software such as polymorphism, inheritance and dynamic binding, which complicate maintenance tasks.[WILD92] They maintain that the unrestricted use of these features tends to drive maintenance costs up by increasing the complexity and decreasing the transparency of the software.

These and the many other studies on the relationship of software product size, complexity and quality to maintenance costs must be considered when developing a maintenance costing model. This is, however, only the product side. There is also the process side to be considered, i.e. which features of the maintenance environment such as source accessability, browsing, editing, compiling and testing facilities, communication facilities, influence maintenance costs, etc. Here, studies have been made by Thadhani [THAD84], Lambert [LAMB84], and Butterline [BUTT92], which show that maintenance productivity can be increased by up to 37% by improvements to the maintenance environment, in particular the use of a dedicated maintenance workbench as proposed by this author in another paper.[SNEE95B]

All of this goes to show how difficult it is to create a comprehensive model for estimating software maintenance costs. It is not only necessary to consider the product to be maintained, but also the environment in which it is to be maintained. Finally, the quality of the maintenance personnel has to be considered. These are the three general cost drivers in software maintenance as depicted in Figure 1.

3. THE SOFTCALC MODEL FOR ESTIMATING MAINTENANCE COSTS

In an attempt to synthesize and operationalize the many research results cited above and to merge them with conventional estimating techniques, the author's company has developed a tool-supported process for estimating maintenance tasks. The process is carried out in seven steps. (See Figure 2.)

In step 1, an automated audit of the software is conducted to determine its size, complexity, and quality.

In step 2, the impact domain of the software affected by the planned maintenance action is circumscribed.

In step 3, the size of the impact domain is measured in two or more of the following size metrics

- lines of code,
- statements,
- function-points,
- data-points,
- object-points.

In step 4, the size measure is adjusted by a complexity factor calculated using a code auditor.

In step 5, the size measure is adjusted by the external and internal quality factors. The latter which reflects the software maintainability is also obtained automatically from a code audit.

In step 6, the size measure is adjusted by a productivity influence factor depending on the estimation method used.

In step 7, the adjusted size measure is transposed into maintenance effort by means of a productivity table.

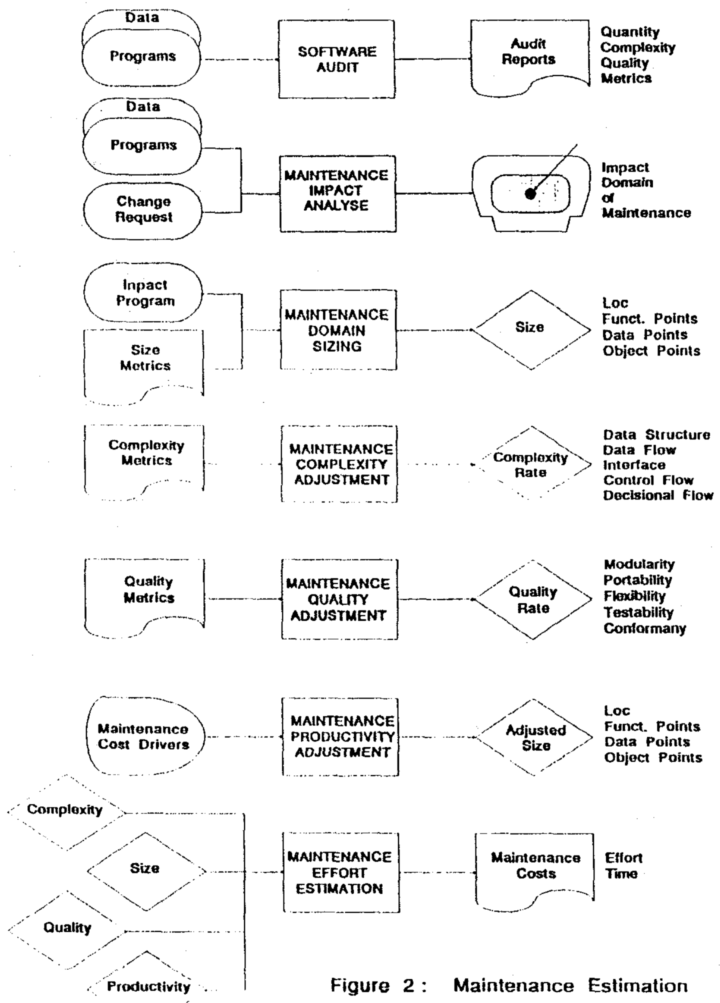

Figure 2 : Maintenance Estimation Process

4. MEASURING THE SIZE OF SOFTWARE

There are a number of different measurements of software size

- lines of code,
- statements,
- function-points,
- data-points, and
- object-points,

each of which has advantages and disadvantages depending on the nature of the software.

When dealing with software written in a third generation procedural language such as FORTRAN, COBOL, or PL/1, lines of code or statements can be used to measure size. The problem with counting lines is that a statement may be spread out over several lines. The number of lines is often a question of programming style. Therefore, it is better to use statements especially with a language like COBOL. Each data definition and each verb denotes a statement. This way ON and END lines are not counted. The same applies to PL/1, where each statement is terminated by a semicolon. Only in FORTRAN is there really a relatively fixed relation between statements and lines of code. The only thing which speaks for lines of code is that they are easy to count. However, with a simple tool it is just as easy to count statements. In any case, when dealing with existing software, it is recommended to count statements.

In the case of software systems written in a 4th generation language such as FOCUS, ADS-ONLINE, ABAP, or NATURAL, it is best to measure the size of the existing system in data-points. Data-points are a measure for the number of data elements processed, the number of user views, the number of data fields in the views, and the complexity of the interfaces.[SNEE90] Data elements and data fields are weighted with 1, relational tables with 4, database accesses with 2, and user views with 4. User views are adjusted by a complexity factor of 0.75 to 1.25. The sum of the adjusted data-points is a reliable indicator for the size of 4GL systems which connect user interfaces to relational databases.

For object software, the most appropriate measure of size is the object-point. Object-points are derived from the class structures, the messages and the processes or use cases.[SNEE95A]

```
Class-Points =
     (number of class attributes x 1)
  +  (number of class relationships x 2)
  +  (number of class methods x 3)
for each class.

Message-Points =
     [(number of parameters x 1)
  +  (number of senders and receivers x 2)]
  x  complexity rate for each message.

Process-Points =
     (process multiplication factor of 1 to
      6 depending on the process type -
      batch, online, system, realtime)
  +  (number of process variations x 2)
  x  (complexity rate for each process).
```

In the case of classes, the number of class-points for each class is adjusted by the reusability rate. It is important to note that inherited attributes and methods are only counted once, at the base or super class level.
Class-points and message-points can be derived automatically from the code itself. Process-points must be counted in the user documentation. The object-point count is the sum of the class-points, message-points, and process-points. As such it is a union of internal and external views of the software.

Function-Points are a universal measurement of software size for all kinds of systems. They are obtained by counting system inputs, system outputs, system queries, databases, and import/export files.[ALBR83] System inputs are weighted from 3 to 6, system outputs are weighted from 4 to 7, system queries are weighted from 3 to 6, databases are weighted from 7 to 15, and import/export files are weighted from 7 to 10. It is difficult to obtain the Function-Point Count by analyzing the code. The question is what is an input and an output. Counting READ's and WRITE's, SEND's and RECEIVE's, DISPLAY's and ACCEPT's is misleading, since there is no direct relationship between program I/O operations and system inputs and outputs. The actual number of system inputs and outputs is a subset of the program inputs and outputs. This subset has to be selected manually from the superset derived automatically by comparison with the system documentation. The database Function-Point Count can be derived by counting physical databases or relational tables in the database schema. Import/Export Files can be found in the Interface Documentation. In light of the strong reliance on documentation rather than code, there is much more human effort involved in counting Function-Points, unless this count can be obtained from a CASE-Repository.

Independently of what unit of measure is used, the result of the system sizing task is an absolute metric representative of the size of the system as a whole. The next step is to determine what portion of this size measurement is affected by the planned maintenance operation. This is the goal in impact analysis.

5. DETERMINING THE SIZE OF THE IMPACT DOMAIN

Once the size of the system under maintenance has been quantified, it is only necessary to update the size figures after each new release. Maintenance action requests come in the form of error reports and change requests as well as in the form of reengineering project proposals. These requests can be placed in the traditional categories of

- corrective,
- adaptive,
- perfective, and
- preventive

maintenance. [LIEN81]

In order to estimate the costs of a maintenance action, it is first necessary to determine the size of the impact domain of that action, i.e. the proportion of the entire software affected. This study is referred to as impact analysis and has been covered in the literature by Robert Arnold [ARNO89], Malcom Munroe [MUNR94], and others. The object of the impact analysis depends on how the software has been sized. If it is sized in function-points, it will be necessary to analyze the system documentation with the following questions in mind:

- What inputs and outputs must be added?
- What inputs and outputs must be altered?
- What queries must be added?
- What queries must be altered?
- What import/export files must be added?
- What import/export files must be altered?
- What files or databases must be added?
- What files or databases must be altered?

If inputs, outputs, queries, import/export files, or databases are to be added, then their function-points must be counted in full. If they are to be altered, then the question is to what extend. Here, the analyst must exercise judgement in coming up with a percent change factor. One method is to count the number of data elements to be changed or added relative to the sum of the data elements in that particular input/output interface or file. (See Figure 3)

If the system is sized in object-points, it is the goal of the analysis to first establish which classes have to be added or altered. If a class is to be added, then its attributes, relationships, and methods are counted in full. If it is to be altered, then only those attributes, relationships, and methods which are affected by the change are counted. The same applies to

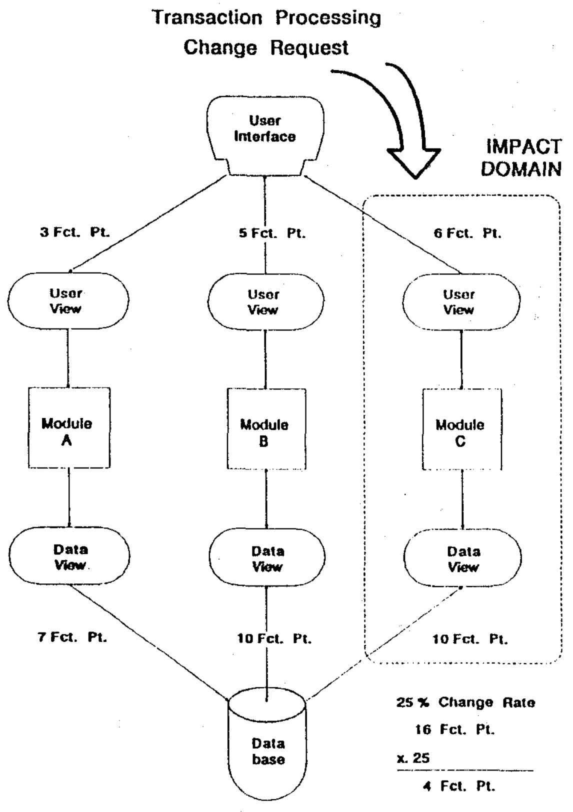

Figure 3 : Sizing the Impact Domain

messages. New messages are counted in full, altered messages are counted relative to the proportion of parameters, senders and receivers affected. Finally, new processes are counted in full, but altered processes are only counted if there is a new variant to be implemented. The sum of the impacted object-points is then some portion of the total number of object-points. This will be the most likely method of sizing maintenance actions on object-oriented software.

In case the system has been sized in data-points, the focus of the impact analysis is on the data model. Here, it is required to answer the following questions:

- What data entities must be added?
- What data elements in existing data entities must be added or changed?
- What relationships between existing data entities must be added or changed?
- What user views must be added?
- What data fields in existing user views must be added or changed?
- What relations between existing data entities and existing user views must be added or altered?

New entities and user views are counted in full. Altered entities and user views are counted in proportion to the number of their attributes and relationships affected by the planned maintenance action. The result is some portion of the total system size in Data-Points.

Finally, existing systems can always be sized in terms of lines of code or statements. The object of the impact analysis is, in this case, the code. The problem is to determine first what components are affected by the change and then to determine what percentage of the statements, i.e. data declarations and procedural instructions, are affected. Of all the sizing approaches used in impact analysis this is surprisingly the most difficult, because of the fine granularity. Either one uses some kind of sophisticated slicing technique to identify precisely which program paths and which data must be altered, or one has to make an educated guess. The former method is accurate but costly, the latter cheap but inaccurate. Thus, counting statements affected by maintenance is no easy task. It may be better to size maintenance impact domains in Function-Points, Object-Points, or Data-Points, and then to transpose this count into a statement count based on a size relationship factor as that published by Capers Jones.[JONE91]

6. ADJUSTING THE SIZE OF THE MAINTENANCE IMPACT DOMAIN BY THE COMPLEXITY FACTOR

From the voluminous amount of literature on the subject it is obvious that the complexity of the software has a definite effect on the maintenance effort. The question is what complexity metrics to use. In his book on Software Complexity, Horst Zuse describes no less than 139 different complexity metrics.[ZUSE90] Therefore, anyone faced with the task of assessing complexity must somehow make a selection.

In the development of its audit tools for Assembler, PL/1, COBOL, and NATURAL, the author's company has reduced the many possible metrics to five basic ones:

- data complexity,
- data usage complexity,
- control flow complexity,
- decisional complexity, and
- interface complexity.

Data Complexity is derived using Chapin's Q-complexity measure.[CHAP79] Data Usage Complexity is based on Elshof's referential frequency complexity.[ELSH76] Control Flow Complexity is McCabe's cyclomatic complexity.[MCCA76] Decisional Complexity is an adaptation of McClure's decisional nesting metric.[MCCL81] Interface Complexity is based on Henry and Kafura's system cohesion metrics.[HENR81] All of the metrics are converted onto a rational scale of 0 to 1 with 0.5 being the median complexity, 1 the highest acceptable, and 0 the lowest possible. The resulting coefficient is then added to 0.5 the median complexity to give a multiplication factor with a base of 1. This factor is used to adjust the size of the impacted software by complexity. Thus, a complexity factor of 1.1 will cause a 10% increase in the size of the impact domain.

Complexity-Adjusted-Size =

Raw-Size * Complexity-Factor

7. ADJUSTING THE SIZE OF THE MAINTENANCE IMPACT DOMAIN BY THE QUALITY FACTOR

The quality of software has a definite influence on productivity, not only in development, where it is an objective to be achieved, but also in maintenance, where it is not only an objective to be achieved, but also a constraint to be overcome. In development, quality is definitely a cost driver. The higher the quality goals, the higher the costs. In maintenance, this is not so simple. On the one hand, quality in terms of reliability, security, performance, usability, etc., must be maintained at the same level if not enhanced, otherwise maintenance degradation sets in. On the other hand, quality in terms of portability, flexibility, readability, modularity, interoperability, etc. is a constraint. If the quality of the existing software is low, the costs of implementing maintenance actions will be higher. If it is high, then these costs will be lower.[BOEH84]

For this reason, the quality factor has to be computed twice. First, the external quality characteristics - reliability, security, integrity, usability, time efficiency and space efficiency, are rated on a scale from 0.5 to 1.5, whereby 1 is the industry median. The highest possible quality is 1.5, the lowest acceptable is 0.5. The system external quality factor is the average of the individual external quality factors.

In a second step, the internal quality characteristics - portability, flexibility, readability, modularity, data independence, and interoperability are rated on a similar scale to the program complexity with a lower bound of 0, an upper bound of 1, and a median of 0.5. The average of these tool-derived internal metrics is then subtracted from 1, i.e. inverted, and added to 0.5 to give a multiplication factor as expressed in the rule

MF = (1 - Average-Quality) + 0.5

In this way, an average quality above 0.5 will reduce the size of the impact area, whereas an below average quality will increase the size. The problem here is, of course, in defining what is an average grade for maintainability, i.e. what is an average cyclomatic complexity, an average nesting level or an average number of interfaces.

The size measure in statements, data-points, object-points, or function-points is adjusted by multiplying it first by the external quality factor and then by the internal quality factor, both being independent of one another. For example, if the function-point count of the software impacted is 20, the external quality factor 1.1, and the inverted internal quality factor 0.8, the adjusted function-point count will be:

20 * 1.1 * 0.8 = 17

8. ADJUSTING THE SIZE OF THE MAINTENANCE COSTS BY THE PROJECT INFLUENCE FACTOR

The project influence factor is computed differently by each estimation method. In the COCOMO method it is the product of 16 project cost drivers on a scale of 0.4 to 2.0. The function-point method weighs 14 factors on a scale of 0 to 5 and adds the sum multiplied by 0.01 to 0.65.[LOWJ90] The data-point method weighs 10 factors on a scale of 1 to 5 and subtracts the sum multiplied by 0.01 from 1.25. The object-point method assigns weights from 0 to 4 to 10 different factors and subtracts the sum multiplied by 0.01 from 1, meaning that the number of object-points can only be adjusted downwards.

Multiplying the system size by the project influence factor renders the final adjusted size of the impact domain domain.

Adjusted-Size =

Raw-Size * Complexity-Factor * Quality-Factor * Influence-Factor.

9. COMPUTING THE MAINTENANCE EFFORT

The relationship between the adjusted size of the impact domain and the effort required to perform the maintenance action is derived from a maintenance productivity table. In general, it can be said that maintenance productivity is about half of development productivity. If the development productivity is 50 statements per day, then the maintenance productivity will be 25. Likewise, a function-point productivity of 10 function-points per developer man-month means 5 function-points per maintenance man-month.

One reason for this is the additional effort required to comprehend the software. A second reason is the additional effort needed to integrate and retest the altered components. Together, these factors drive the effort up by some 100%. This conclusion has been made by several independent studies of maintenance productivity.[GRAD87]

Essentially, it is up to each software maintenance shop to calibrate their own maintenance productivity curves reflected in the tables based on their own experience. This will make the estimating process more accurate.

10. A CASE STUDY IN ESTIMATING THE COSTS OF ALTERING REENGINEERED COBOL PROGRAMS

In the fall of 1993, the author's company had just completed the reengineering of nine Basic Assembler programs with 7,500 statements in a financial application system to COBOL-85. The new COBOL versions were due to go into production by the end of the year. However, due to changes in the tax laws the programs had to be altered before going into production. The changes were first built by the customer programmers into the original Assembler programs and tested. Fortunately, they were also marked, so it was possible to identify them.

After the original Assembler programs had been validated, the statements in the COBOL programs which corresponded to the marked Assembler statements were identified. The changes were significant so that in all some 372 Assembler statements or 5% of the total code had been altered. This corresponded to precisely 410 COBOL statements.

The complexity measurements of the Assembler code derived by the ASMAUDIT tool were:

Data Complexity	= 0.743
Data Usage Complexity	= 0.892
Control Flow Complexity	= 0.617
Decisional Complexity	= 0.509
Interface Complexity	= 0.421

giving an average complexity for the Assembler programs of 0.636. This was transposed to a complexity factor of 1.14. (See Figure 4.)

The complexity measurements of the COBOL code derived by the COBAUDIT tool were:

Data Complexity	= 0.744
Data Usage Complexity	= 0.811
Control Flow Complexity	= 0.529
Decisional Complexity	= 0.506
Interface Complexity	= 0.421

giving an average complexity for the COBOL programs of 0.602. This was transposed to a complexity factor of 1.10. (See Figure 5.)

```
+------------------------------------------------------------------------+
!                                                                        !
!                         S Y S T E M                                    !
!                                                                        !
!                  S U M M A R Y    R E P O R T                          !
!                                                                        !
!  PROGRAM LANGUAGE:   370-Assembler                  DATE:  11.10.93    !
!  SYSTEM NAME:        Coupons                        PAGE:  001         !
+------------------------------------------------------------------------+
!                 Q U A N T I T Y      M E T R I C S                     !
+------------------------------------------------------------------------+
!                 D A T A   S I Z E   M E T R I C S                      !
!                                                                        !
!         NUMBER OF FILES/DATABASES            ==========>     11        !
!         NUMBER OF DATA OBJECTS               ==========>     18        !
!         NUMBER OF DATA DECLARED              ==========>   2473        !
!         NUMBER OF DATA REFERENCES            ==========>   1805        !
!         NUMBER OF ARGUMENTS                  ==========>   3008        !
!         NUMBER OF RESULTS                    ==========>   2632        !
!         NUMBER OF PREDICATES                 ==========>   1428        !
!         NUMBER OF COPY/MACROS                ==========>     38        !
!         NUMBER OF DATA-POINTS                ==========>    712        !
!                                                                        !
!                 P R O C E D U R E   S I Z E   M E T R I C S            !
!                                                                        !
!         NUMBER OF STATEMENTS                 ==========>   7521        !
!         NUMBER OF MODULES (CSECTS)           ==========>      9        !
!         NUMBER OF BRANCHES                   ==========>   2933        !
!         NUMBER OF GO TO BRANCHES             ==========>   1254        !
!         NUMBER OF SUBROUTINE CALLS (BAL)     ==========>    207        !
!         NUMBER OF MODULE CALLS (BALR)        ==========>     19        !
!         NUMBER OF DATA REFERENCES            ==========>   4961        !
!         NUMBER OF FUNCTION-POINTS            ==========>     55        !
!         NUMBER OF LINES OF CODE              ==========>   9994        !
!         NUMBER OF COMMENT LINES              ==========>   1093        !
+------------------------------------------------------------------------+
!                 C O M P L E X I T Y    M E T R I C S                   !
+------------------------------------------------------------------------+
!         DATA COMPLEXITY                      ==========>  0,743        !
!         DATA FLOW COMPLEXITY                 ==========>  0,892        !
!         INTERFACE COMPLEXITY                 ==========>  0,421        !
!         CONTROL FLOW COMPLEXITY              ==========>  0,617        !
!         DECISIONAL COMPLEXITY                ==========>  0,509        !
+------------------------------------------------------------------------+
!         INTERNAL COMPLEXITY                  ==========>  0,636        !
+------------------------------------------------------------------------+
!                 Q U A L I T Y       M E T R I C S                      !
+------------------------------------------------------------------------+
!         MODULARITY                           ==========>  0,339        !
!         PORTABILITITY                        ==========>  0,214        !
!         MAINTAINABILITY                      ==========>  0,364        !
!         TESTABILITY                          ==========>  0,472        !
!         CONFORMITY                           ==========>  0,702        !
+------------------------------------------------------------------------+
!         INTERNAL QUALITY                     ==========>  0,418        !
+------------------------------------------------------------------------+
```

Figure 4: Assembler System Metrics

Adjusting the size of the Assembler Impact Domain gave a statement count of 424. Adjusting the size of the COBOL Impact Domain gave a statement count of 451, which was still greater than that of the Assembler due to the additional statements created by the automatic conversion.

The external quality of the code remained constant at 0.05. However, the internal quality had risen as a result of the reengineering from a previous 0.418 for the Assembler to 0.473 for the COBOL version. When inverted, this gave the multiplication factor of 1.08 for Assembler and 1.03 for COBOL. This resulted in a quality adjusted COBOL statement count of 415, as opposed to the quality adjusted Assembler statement count of 458. At this point, the size of the COBOL impact domain was still larger.

The project influence factors had also improved as a result of the reengineering. It was now possible to edit, compile, and test the programs on a PC-workstation. The product of the cost drivers was 1.26 for the Assembler Code. For the COBOL Code it was 0.98.

Thus, whereas the final adjusted Assembler statement count came out to be 577, the adjusted COBOL statement count was 21% less at 456.

At a productivity rate of 20 Assembler statements or 30 adjusted statements per man-day, it took some 19 man-days to adapt and retest the original Assembler programs. On the COBOL side it took only 10 man-days to duplicate the changes and retest the programs giving a maintenance productivity rate of 45 adjusted statements or 41 real statements per man-day. The effort of specifying the changes was not included on either side, but it would have been the same for both.

This is a good example of how reengineering can reduce maintenance costs. Even though the automatic conversion of Assembler to COBOL resulted in more code, it was still possible to alter the COBOL code quicker and cheaper than the original Assembler code. This was due primarily to the better support offered by the Microfocus COBOL Workbench which underlies the impact of the environment on maintenance productivity.

11. CONCLUSIONS

There is a definite need to extend current software cost estimation methods which are focused primarily on the development process to the maintenance process. In principle, all of the methods can be applied to predicting maintenance effort. What is necessary is an adequate means of defining the impact domain of planned alterations. Once this has been solved, the scope of the maintenance action can be measured. What is also needed are reliable metrics to assess the quality of the software product. ISO-9126 is a good step in the right direction.[ISO94] With the help of the quality metrics, the size can be adjusted. Equally important as the product metrics are the process metrics for weighting productivity. Finally, what is needed are empirically

```
+----------------------------------------------------------------------+
!                                                                      !
!                          S Y S T E M                                 !
!                                                                      !
!                    S U M M A R Y    R E P O R T                      !
!                                                                      !
! PROGRAM LANGUAGE:   COBOL-85                          DATE:  12.10.93 !
! SYSTEM NAME:        Coupons                           PAGE:  001      !
+----------------------------------------------------------------------+
!              Q U A N T I T Y       M E T R I C S                     !
+----------------------------------------------------------------------+
!              D A T A   S I Z E   M E T R I C S                       !
!                                                                      !
!         NUMBER OF FILES/DATABASES          ==========>     11        !
!         NUMBER OF DATA OBJECTS             ==========>     18        !
!         NUMBER OF DATA DECLARED            ==========>   2488        !
!         NUMBER OF DATA REFERENCES          ==========>   1827        !
!         NUMBER OF ARGUMENTS                ==========>   3023        !
!         NUMBER OF RESULTS                  ==========>   2632        !
!         NUMBER OF PREDICATES               ==========>   1428        !
!         NUMBER OF COPY/MACROS              ==========>     38        !
!         NUMBER OF DATA-POINTS              ==========>    712        !
!                                                                      !
!              P R O C E D U R E   S I Z E   M E T R I C S             !
!                                                                      !
!         NUMBER OF STATEMENTS               ==========>   8271        !
!         NUMBER OF MODULES (SECTIONS)       ==========>     39        !
!         NUMBER OF BRANCHES                 ==========>   2673        !
!         NUMBER OF GO TO BRANCHES           ==========>   1142        !
!         NUMBER OF PERFORMS                 ==========>    357        !
!         NUMBER OF MODULE CALLS             ==========>     19        !
!         NUMBER OF DATA REFERENCES          ==========>   5174        !
!         NUMBER OF FUNCTION-POINTS          ==========>     55        !
!         NUMBER OF LINES OF CODE            ==========>  10759        !
!         NUMBER OF COMMENT LINES            ==========>   1579        !
+----------------------------------------------------------------------+
!              C O M P L E X I T Y   M E T R I C S                     !
+----------------------------------------------------------------------+
!         DATA COMPLEXITY                    ==========>  0,744        !
!         DATA FLOW COMPLEXITY               ==========>  0,811        !
!         INTERFACE COMPLEXITY               ==========>  0,529        !
!         CONTROL FLOW COMPLEXITY            ==========>  0,506        !
!         DECISIONAL COMPLEXITY              ==========>  0,421        !
+----------------------------------------------------------------------+
!         INTERNAL COMPLEXITY                ==========>  0,602        !
+----------------------------------------------------------------------+
!              Q U A L I T Y      M E T R I C S                        !
+----------------------------------------------------------------------+
!         MODULARITY                         ==========>  0,396        !
!         PORTABILITITY                      ==========>  0,389        !
!         MAINTAINABILITY                    ==========>  0,398        !
!         TESTABILITY                        ==========>  0,467        !
!         CONFORMITY                         ==========>  0,715        !
+----------------------------------------------------------------------+
!         INTERNAL QUALITY                   ==========>  0,473        !
+----------------------------------------------------------------------+
```

Figure 5: COBOL System Metrics

founded correlations between software size, complexity, and quality on the one side and maintenance effort on the other.
Only when we are better able to predict the costs of fulfilling maintenance requests, will it be possible to control the software maintenance process.

REFERENCES

ALBR83 Albrecht, A.J./Gaffney, J.E.: "Software Function, Source Lines of Code, and Development Effort Prediction - A Software Science Validation" in IEEE Trans. on S.E., Vol. SE-9, No. 6, Nov. 1983, p.639

ARNO89 Arnold, R.: "Software Impact Analysis", Tutorial at IEEE Conference on Software Maintenance, Orlando, Fla., 1989

BOEH81 Boehm, B.: "Software Engineering Economics", Prentice-Hall, Englewood Cliffs, N.J., 1981

BOEH83 Boehm, B.: "Economics of Software Maintenance", in Proc. of Software Maintenance Workshop, IEEE Press, Monterey, Cal., 1983, p.9

BOEH84 Boehm, B.: "Software Engineering Economics", IEEE Trans. on S.E., Vol. SE-10, No. 1, Jan. 1984, p.4

BOEH88 Boehm, B.: "Understanding and Controlling Software Costs" in IEEE Trans. on S.E., Vol. 14, No. 10, Oct. 1988

BUTT92 Butterline, M.A.: "From Mainframe to Workstations - Offloading Application Development", QED Publishing Group, Boston, 1992, p.59

CARD87 Card, D./Cotnoir, D./Goorevich, C.: "Managing Software Maintenance Cost and Quality" in Proc. of Conference on Software Maintenance, IEEE Press, Austin Tex., 1987, p.145

CHAP79 Chapin, N.: "A Measure of Software Complexity" in Proc. of Natural Computer Conf., Chicago, 1979

COLE94 Coleman, D./Ash, D./Lowther, B./Oman, P.: "Using Metrics to Evaluate Software System Maintainability", IEEE Computer, August 1994, p.44

ELSH76 Elshof, J.: "An Analysis of Commercial PL/1 Programs", IEEE Trans. on S.E., Vol. SE-2, No. 3, June 1976

GRAD87 Grady, Robert: "Measuring and Managing Software Maintenance" in IEEE Software, Vol. 4, No. 9, Sept. 1987, p. 35-45

HENR81 Henry, S./Kafura, D.: "Software Structure Metrics based on Information Flow", IEEE Trans. on S.E., Vol. SE-7, Sept. 1981

ISO94 International Standard Organization: ISO-9126 "Software Product Quality Characteristics", Geneva, 1994

JONE91 Jones, C.: "Applied Software Measurement", McGraw-Hill, New York, 1991

KEME94 Kemerer, C./Chidamber, S.: "A Metrics Suite for Object-Oriented Design", IEEE Trans. on S.E., Vol. 20, No. 6, June 1994, p.476

LAMB84 Lambert, G.: "A Comparative Study of System Response Time on Programmer Developer Productivity", IBM Systems Journal, Vol. 23, No. 1, 1984, p.36

LANN94 Lanning, D./Khoshgoftaar, T.: "Modeling the Relationship between Source Code Complexity and Maintenance Difficulty" in IEEE Computer, Sept. 1994, p.35

LIEN81 Lientz, B./Swanson, E.B.: "Problems in Application Software Maintenance", Comm. of ACM, Vol. 24, No. 11, Nov. 1981, p.763

LOWJ90 Low, G./Jeffery, D.R.: "Function-Points in the Estimation and Evaluation of the Software Process", IEEE Trans. on S.E., Vol. 16, No. 1, Jan. 1990, p.64

MCCA76 McCabe, T.: "A Complexity Metric" in IEEE Trans. on S.E., Vol. 2, No. 4, Dec. 1976, p.308

MCCL81 McClure, C.: "Managing Software Development and Maintenance", Van Nostrand Reinhold, New York, 1981

MUNR94 Munroe, M./Turver, R.: "Early Impact Analysis Technique for Software Maintenance", Journal of Software Maintenance, Vol. 6, No. 1, Jan. 1994, p.35

ROMB87 Rombach, D.: "A Controlled Experiment on the Impact of Software Structure on Maintainability", IEEE Trans. on S.E., Vol. SE-13, No. 3, March 1987, p.344

SNEE90 Sneed, H.: "Die Data-Point-Methode" in ONLINE, ZfD, No. 5/90, May 1990, p.48

SNEE95A Sneed, H.: "Estimating the Costs of Object-Oriented Software", Proc. of Software Cost Estimation Seminar, Systems Engineering Ltd., Durham, G.B., March 1995

SNEE95B Sneed, H.: "Implementation of a Software Maintenance Workbench at the Union Bank of Switzerland" in Proc. of Oracle Conference on System Downsizing, Oracle Institute, Munich, April 1995

THAD84 Thadhani, A.: "Factors Affecting Programmer Productivity during Application Development" IBM Systems Journal, Vol. 23, No. 1, 1984, p.19

VESS83 Vessey, I./Weber, R.: "Some Factors Affecting Program Repair Maintenance: An Empirical Study" Comm. of ACM, Vol. 26, No. 2, Feb. 1983, p.128

WILD92 Wilde, N./Huitt, R.: "Maintenance Support for Object-Oriented Programs", IEEE Trans. on S.E., Vol. 18, No. 12, Dec.. 1992, p.1038

ZUSE90 Zuse, H.: "Software Complexity - Measure and Methods, De Gruyter Verlag, Berlin, 1990