A Cost Model for Software Maintenance & Evolution

Harry M. Sneed
Anecon GmbH, Vienna Austria
Institut für Wirtschaftsinformatik, University of Regensburg, Bavaria
Email: Harry.Sneed@T-Online.de

Abstract: *The purpose of this essay is to present a costing model for software maintenance and evolution based on a separation of fixed and variable costs. There has always been a problem in distinguishing between the maintenance activities covered by the standard maintenance fee and those charged extra to the user. Separating these two types of costs is essential to every maintenance operation to prevent costs from getting out of control. In this paper the author proposes a solution, which can lead to better cost estimations and a financially more stable maintenance operation. Particular emphasis is placed on a sharp division between work done to maintain the system functionality as it is and work done to enhance that functionality.*

Keywords: *Software Maintenance and Evolution, Software Product Management, Software Life Cycle Costing Models, Maintenance Cost Estimation.*

Rationale for a Maintenance Cost Model

There are many costing models for estimating software development, including Cocomo [1], Function-Point [2], Object-Point [3] and Slim [4], among others. There are, however, few models which deal specifically with maintenance as such. These tend to treat maintenance as a continuation of development on a reduced scale, calculating maintenance costs based on the costs of the original development, adjusted by some influence factors.

Recent studies have shown however that the costs of maintaining a system may not be related to the costs of development. In a survey by Reifer, Boehm and Basili of application systems developed using components off the shelf, the costs of maintaining these systems came out to be some 40% more than the costs of maintaining systems developed from scratch, although the development of the latter systems was much more expensive. [5] Another study of agile projects conducted in Europe showed that systems which cost less to develop are more likely to cost more to maintain. [6] So basing the prediction of maintenance costs on the costs of development seems to no longer hold true, if in fact it ever did.

Another questionable approach is to try and predict maintenance costs based solely on characteristics of the software. This approach is very popular among researchers working in the metric field, since software can be measured and the metrics related to effort in correcting or changing the code. Some researchers have attempted to correlate code complexity with effort [7], some have tried to correlate coupling with effort [8] and others have examined the relationship between architectural quality and maintenance costs. [9]. All of this research is useful in helping to identify cost drivers but it is much too limited in scope.

Calculating maintenance costs is a multi-dimensional problem and the software itself is only one of the many dimensions of that problem. There is not only a product to be maintained, but also a maintenance process, a maintenance environment, maintenance personnel and the tools available. All of these factors are equally as important to predicting maintenance costs as the software itself. Therefore, it is a grave misrepresentation of the problem to imply that maintenance costs are mainly determined by the nature of the software. There may be some correlation in a local laboratory experiment where all of the other influencing factors can be neutralized, but it will never hold up in real projects. There are simply too many other factors to be considered.

The purpose of this paper is to expose this myth and instead to propose a multidimensional costing model based on data collected from the maintenance process as well as from the product being maintained. The model also enforces a strict division between standard maintenance, defined as error correction and system adaptation, and evolution, defined as functional enhancement and technical improvement. The costs for these different activities have to be estimated using different approaches.

Covering Maintenance Costs

A reoccurring topic in software engineering is the separation of maintenance activities from ongoing development. It may not be a topic for the programmer who is doing both simultaneously, but it is definitely a problem for those who have to pay his salary. Normally, maintenance costs are covered by an annual maintenance fee, which is a certain

percentage of the original development costs or of the purchasing price. A typical maintenance fee is 15% of the development costs. Users would like to believe that this gives them the right to ask for whatever they deem necessary to make their work easier. The party responsible for the maintenance takes another standpoint. It will contend that that the maintenance fee only covers the costs of error correction and essential adaptations required to keep the system in operation. All else should be charged extra, including optimizations, renovations, enhancements and non essential adaptations.

These conflicting views of what can be expected from a maintenance operation often burden the relationship between the users and the maintenance organization. Where the maintenance organization is too strict in charging user requests, users become frustrated and dissatisfied. Where the maintenance organization is too lenient in accepting requests as being covered by the maintenance fee, costs tend to get out of hand. In the former case, users may look for another alternative. In the latter case, the maintenance party will operate at a loss, unless it has another users to cover the additional costs. This inherent conflict of interest becomes particularly acute in the case where the user organization is dependent upon an outside party to provide it maintenance support.

The situation has become further confused by the advent of iterative and cyclic development. [10] Here maintenance and development have become highly intertwined. Developers are maintaining the last version while developing the next one. It is difficult enough to keep the tasks apart, much less to separate the costs. However, without doing so, it is impossible to keep costs from going out of control. Costs must be accounted for and charged either against the current version in production or against the new version under development.

There is, therefore, a definite need to resolve this issue from the beginning by clearly defining what is covered by the maintenance fee and what is not. On the one hand, this is a legal issue to be settled in the service level agreement. On the other hand, it is a technical issue, since the definition of what is to be covered by a fixed rate and what is to be charged extra, is in fact a question of the difference between maintenance and evolution. Maintenance costs should be covered by the maintenance fee whereas evolution costs need to be negotiated upon demand. This separation precludes that these two notions are well defined and understood by both parties.
Unfortunately, that is not the case. Even among the members of the academic community, pursuing research in the field of software maintenance, there is no real consensus as to what distinguishes maintenance from evolution. In fact, many researchers use the two notions interchangeably, as can be often detected in the Journal of Software Maintenance and Evolution.[11] This bad practice prevails despite the efforts of some researchers to draw a line between the two. Although this distinction may seem unnecessary and academic to developers and researchers, it is of utmost importance to managers responsible for the budget and legal advisors responsible for the maintenance contracts. They must be able to precisely define who pays for what. Furthermore, the separation of maintenance from evolution is essential to any maintenance costing model.

Software Maintenance ?

The confusion on the issue of what is maintenance as opposed to what is development goes back to the beginning of electronic data processing. In an early publication of the American General Accounting Office – GAO – maintenance was proclaimed to be all work done on a system after it goes into production the first time.[12] For the accountants this was a clear division. They had a cutoff date, the date of initial delivery. All costs prior to that date were charged to the development, all costs after that date were charged to maintenance. Development costs were project costs and had to be covered by the project budget. Maintenance costs were operation costs and were covered by the annual DP operations budget.

Lientz and Swanson did not explicitly refer to this governmental decree, but they seemed to have accepted the accountant's definition when building up their maintenance categories of

- corrective,
- adaptive and
- perfective maintenance

to include all types of work done on a software system after its initial release. Perfective maintenance was meant to include both optimization, renovation, i.e. reengineering, and enhancement. In their original survey of 457 user organizations perfective maintenance made up for circa 42% of the total costs occurred after the initial release date. This showed that there was still a lot of development being charged to the maintenance budget.[13]

Throughout the 1980s this definition of maintenance as being everything done after the system first went into operation was widely accepted. In the IEEE Life Cycle Model of 1980, maintenance was seen as the final phase of the life cycle where errors were corrected and minor

improvements made. [14] As it later turned out this phase was shown to account for more than twice as much as the previous phases combined, yet the adherence to the waterfall model prevailed until the late 1980s when Boehm published his article on the spiral model of software development. [15]

Still in the early 1990's, all work done on a system after the initial release was considered maintenance as shown in the Nosek and Palvia study which was a repeat of the Lientz and Swanson survey using the same categories.[16] This definition of maintenance was supported by the early ICSM conferences and by the Durham European Workshop on Software Maintenance as can be concluded from the papers published at that time. One reason for that was the intricate relationship between such maintenance activities as reverse engineering, program comprehension, reengineering and impact analysis from the researchers point of view. This view was carried over into several Esprit Projects of that time including ReDo, Macs, Docket and Renaissance[17].

The problem with this early definition of maintenance is that industrial managers have an entirely different viewpoint as to what maintenance consists of. Business managers perceive their IT operations from the point of view of their budgets. To them activities such as reengineering, reverse engineering, repository construction, error correction and functional enhancement represent unique cost categories, which have to be charged to a cost carrier. There are some costs, which can be assigned to a system's operational budget and others, which have to be assigned to a project. Reverse engineering, renovation, migration, integration and functional enhancement are typical projects, which have to be funded and made accountable to some cost center. Error correction, optimization and adaptation are activities, which can be charged to the system operation budget.

From a comptrollers viewpoint reverse engineering and error correction belong to two totally separate worlds that have nothing in common, whereas from a researcher's viewpoint they are strongly related. There is hardly any paper on reverse engineering or reengineering which does not start with the argument that the purpose of this exercise is to make error correction and program changing easier. This relationship may be obvious to researchers, but to a business accountant it is not at all obvious, since error correction costs are overhead costs and the others are project costs.

Software Evolution ?

The notion of software evolution originated from the work of Belady and Lehman at IBM and has since progressed to become a distinct field of research.[18] From years of empirical research Lehman has developed and refined the laws of software evolution which make it possible to predict how complex software systems will evolve over time. Another proponent of the theory of software evolution was Tom Gilb, who propagated evolutionary software engineering even before Boehm's landmark article on the cyclic model.[19] What unites all proponents of the theory of evolution is their common definition of software development as an ongoing, in principle endless process.

Like maintenance evolution is also concerned with what happens to software systems after they go into production, but the focus is different. Error correction is taken for granted or considered part of the continuous improvement process. System change is looked upon as a subset of the enhancement process. The focus is on system growth. Systems are classified according to their growth rate. Systems with a low growth rate, e.g. back office operations running on the mainframe, are termed static. Systems with a high growth rate, e.g. front office operations running on the web, are termed dynamic. [20]

The theory of software evolution has been given a boost by the current practice of extreme programming [21]. What unites the two is their common view of software development as an ongoing process. Since requirements have to be discovered via trail and error, there is no alternative to proceeding one step at a time. In extreme programming, a system is developed in stages, with a project for each stage. The definition of what constitutes the next stage is determined by the current state of knowledge and the time available. In evolutionary development a system is developed one release at a time. What constitutes a release is determined by the pending change requests and the release interval. One might say that evolutionary software development is actually extreme programming in the large.

From a business point of view evolutionary development is an expedition into the unknown. No one can say what the final result will be and what it will cost. This makes planning and budgeting extremely difficult. Understandably, accountants and contractors have a strained relationship to both evolutionary development and extreme programming. The sharp distinction between development and maintenance fitted better to their pursuit of cost calculation and cost control.

The staged Model

Fortunately, Rajlich and Bennett have come up with a compromise solution, which promises to resolve the conflict between maintenance and evolution – the staged model [22]. According to the model, there are five phases in the life cycle of a software system.

- Prototype phase
- Development phase
- Evolution phase
- Maintenance phase
- Retirement phase

Here maintenance and evolution are not competing with one another, but are two distinct periods in the life of a system.

The prototype phase is actually a feasibility study of the costs and benefits of making the system at all. This phase is definitely a case for extreme programming. It has to be budgeted like any research project. A fixed sum is allocated and a deadline arbitrarily set.

With the knowledge obtained from the prototyping phase it is then possible to estimate the costs and time of the development phase with a relative certain degree of accuracy. Here, conventional estimation methods can be employed because the system has been defined. The development phase has a fixed deadline after which the system is released in the state that it is. At this point, the evolution begins.

The evolution phase as a whole cannot be calculated, but each new release can be. For each new release part of the effort will be devoted to correcting the last release and another part to constructing the new release. The new release may contain a number of functional enhancements, it may be the result of a renovation effort, or it may contain only minor adaptations. When the latter type of release prevails, this is a sign that the evolution phase is drawing to an end.

Due to various reasons, i.e. increasing costs and decreasing benefits, the system growth rate will become less as time goes by. At some point in time it will not be worth investing more in the system. It will be frozen. From here on only error corrections and unavoidable changes will be made. This is then maintenance in the restricted sense.

Eventually the system will become obsolete or too expense to maintain. This could be a result of increasing complexity and decreasing quality, but more often it is caused by the loss of knowledge. The personnel who command the technology with which the system was developed and who have the necessary domain knowledge are no longer available. It is also possible that the environment in which the system operates is no longer supported. In some cases, it may be possible to renovate and to migrate the software to another more modern environment. In other cases, it will only be possible to save the data. The software will be replaced. In either case, the existing system ceases to exit.

The cost model presented here is based on the staged model of Rajlich and Bennett. It presupposes an evolution phase in which further development and maintenance are taking place in parallel and that there is a distinct maintenance phase in which only maintenance tasks are taking place. The purpose of the model is to give managers and accountants the possibility of planning and controlling the costs of both maintenance and evolution.

Estimating Maintenance Costs

There are basically two approaches to estimating software change costs. One is by employing impact analysis on a task by task basis, identifying the components affected by the change and calculating the size of the impact domain, which can then be adjusted by the complexity and quality of that domain. This micro-level approach has been described by the author in a previous paper. [23]

The other approach is the macro level approach, which uses data from past releases to predict the costs of correcting and adapting the next planned release. This method of projecting costs based on past performance has been developed and refined by the author for predicting maintenance and evolution costs for a complex financial application system consisting of more than 7 million lines of code, 2,5 million statements and over 180,000 function-points, with a fixed release interval of three months. [24]

The **MainCost** model has a number of parameters which have to be set in order to use the model. These parameters are derived from different sources. These sources are:

- Static Software Analysis
- Dynamic Software Analysis
- Defect Analysis
- Productivity Analysis

From the static analysis of the software , metrics on the size, complexity and quality of the software specifications, code and test cases are collected.

From the dynamic analysis of the software, data is collected on the degree of test coverage.
From the defect analysis, the numbers and types of actual errors are counted and the defect density computed.
From the productivity analysis, data is gathered on the effort booked against error reports, change requests, reengineering tasks and development tasks. Without this data from diverse sources, it is hardly possible to predict maintenance costs. The software itself is not sufficient to build a model on.

Once the data is available, the diverse types of maintenance and evolution tasks can be calculated. The task types covered by the **MainCost** model are:

- Error Correction,
- Routine Change,
- Functional Enhancement and
- Technical Renovation

The premise put forward here is that there is no such thing as a universal approach to estimating maintenance costs. Each activity requires its own estimation method. The objective is to identify the different types of maintenance and evolution tasks and to find an appropriate approach to that particular type of task.

Static Software Analysis

The techniques of static analysis have been well covered in the literature, in particular source code analysis.[25] However, a software system consists of more than just source code. It also encompasses the requirement specifications, the design documents, the test cases and the database schemas. In so far as these semantic artifacts are also maintained and evolved, they should also be objects of the static analysis. For the purpose of cost modeling, the static analysis should measure the size, complexity and quality of the software artifacts being analyzed. There are five different types of artifacts, which have to be analyzed:

- Requirement Specifications
- Design Documents
- Source Code
- Database Schemas
- Test Cases

In the static analysis of the requirement specification, the use cases, functions, business rules, data entities, attributes, user interfaces and system interfaces are counted to derive the quantity of the concept. The complexity of the concept is derived from the relationships between these logical entities. The quality of the requirement specification is a function of its completeness, consistency and formal correctness.

The static analysis of the design documentation is directed toward counting the number of diagrams, their relations to one another and their formal correctness. UML designs are generally stored in a repository, where they can be automatically analyzed. If there is no access to the repository then there are the export files, which can be produced and analyzed. With the advent of model driven development it becomes more and more important to be able to measure the properties of a design and to come up with metrics for its size, complexity and quality. [26]

Source code analysis will provide a number of metrics which have to be aggregated into a single complexity measure, a single quality measure and one or more size measures, i.e. Function-Points, Object-Points, Statements, Locs. It is not the purpose of this paper to describe how this can be done. It has been described elsewhere. What is important is that complexity and quality are rated on a rational scale from 0 to 1 as recommended in the ISO Standard 9126. This normalized scale makes it possible to join the metrics of the source with those of the other artifacts and to adjust the size of the software. [27]

The static analysis of the database schemas will reveal the number of tables and attributes as well as the number of relationships between them. The quality of the database structures can also be assessed as pointed out by Blaha.[28] The results are size, complexity and quality metrics of the data model.

The final target for the static analysis is the test case library. Test cases are now being kept in Excel tables, CSV Files and relational databases. The fact is that test cases have a syntax and a semantic albeit a different one for every environment. Nevertheless, the test cases and their attributes can be counted, their complexity measured and their quality assessed based on the degree of automation and the impact domain. This author has already reported on the results of his research in measuring test cases. [29]

In the case study the design documentation was not maintained, but it was possible to measure the requirement specification, the database, the code and the test cases, which were maintained in a relational database.

Dynamic Software Analysis

The purpose of the dynamic analysis is to measure the test coverage achieved at each release. To this end, either the source code or the byte code has to be instrumented. There are alternative levels of coverage measurement. Some will choose to instrument at the method level, others at the branch level. The level selected should be appropriate to the software under test. It may also be necessary to selectively instrument the code in order to avoid portions of the code, which are not used by this application. [30]

In any case, profile tables will be created at test time showing which instrumented nodes have been traversed how often. Normally there will be a separate table for each component or module. When the test is completed, the tables will be aggregated and a compound coverage rate computed for the system as a whole. The coverage rate indicates which percentage of the relevant code has been executed by the test. This number is important for predicting the expected error rate.

Defect Analysis

During system testing records will be kept of all errors found by the test team. These errors will be categorized by source, type and severity. For this an error tracking system with a build in database is recommended.

During the interval between releases, errors are reported by the users of the system. These customer errors are kept in the same database as the tester errors, but are marked as customer errors. In addition, these errors are assigned to a specific version of a specific release so it is possible to ascertain which errors occurred in what versions of the software. [31]

At the end of each release period, the error database is analysed and the errors summarised by type and by release. Furthermore, errors are divided by the sizes of the source to obtain the error density pro statement, function-point and object-point. These metrics are later used to project the error rate in the next release.

Productivity Analysis

The most important factor in predicting any kind of effort is the productivity factor. It has to be calibrated to the local circumstances. One of the greatest mistakes in software cost estimation is to take over a foreign productivity index, such as the IBM function-point productivity rate, and to apply it to a local estimation. The chance of it fitting is minimal. There are such major differences in environments, tools and people involved, that it is practically impossible to find another similar situation. The best one can hope for is an approximate fit.

By no means should productivity in development be applied to maintenance and evolution. Even between maintenance and evolution there exit significant differences. Generally, it takes longer to locate and correct errors or to insert a change than it does to add a additional method to an existing class or to add an additional class to an existing component. Reengineering code has its own specific productivity curve. The result of this observation is the need to maintain different productivity indexes for different types of tasks. There are separate productivity rates for error correction, change, functional enhancement and renovation.

It is the responsibility of the maintenance organization to enforce a strict time accounting system with which personnel are required to book their hours against specific maintenance and evolution tasks. From the accounting system effort can be derived by task type and related to the code volume, function-points, object-points or other size factors affected by that task type, e.g. the statements changed or the function-points added. The result will be an average productivity rate pro task type and size measure.

Predicting Error Correction Costs

The costs of correcting serious errors, i.e. trouble shooting, are projected from the costs of error correction in the previous release. The following data is required:

- Number of serious errors in the last release
- Average effort per error correction in the last release
- Adjusted size of the last release
- Adjusted size of the next release
- Test coverage of the last release
- Test coverage of the next release

The key questions are what will be the number of errors in the next release and what will be the average effort required to correct them. This will give the effort required for error correction. The number of errors to be expected is projected from the number of errors in the last release adjusted by the difference in test coverage. The effort required to correct them is projected from the previous effort

per error correction adjusted by the difference in the size, complexity and quality of the last and next releases.

Assuming that there is a relation between test coverage and the number of errors not discovered by the test, i.e. those errors reported by the user, then the error density of the last release is the number of errors reported relative to the uncovered portion of that release.

$$Error\_Density = \frac{Errors\_reported}{Size \times (1 - TestCoverage)}$$

With a release of 20,000 statements, a test coverage of 75% and 50 reported errors, the error density for the uncovered portion would be 0.01.

The number of projected errors remaining after the test of the new release will be the uncovered portion of the new release multiplied by the error density of the last release.

$$Number\ of\ expected\ Errors = (New\_Size \times (1 - New\_Test\_Coverage) \times Error\_Density$$

If the new release is 10% larger than the previous one, it will have 22,000 statements. Of this 85% will be covered by the new improved test leaving 3300 statements uncovered. The previous error density of 0.01 applied to this new uncovered portion would indicate that there will be 33 errors remaining in the new release after testing.

The next step is to calculate the effort required to correct these errors. For that the average effort per error in the last release is adjusted by the difference in size, complexity and quality between the last and the next release. The size of each release should be adjusted by the complexity and the quality.

$$Adjusted\_Size = Size \times (Complexity / Quality)$$

Therefore, if the complexity index of the last release with 20,000 statements was 0.48 and the quality index 0.64 then the adjusted size will be 20,000 x 0.75 = 15.000. According to the laws of evolution the software complexity will tend to increase while the quality will tend to decrease. In the new release with 22,000 statements the measured complexity may be 0.52 and the quality 0.60. This would give an adjusted size of 18,920 statements. The difference between the last and the new release is

$$\frac{New\_Size\ or\ 18{,}920}{Last\_Size\ or\ 15{,}000} = 1.26$$

This growth factor is multiplied with the average effort per error of the last release to give the average effort per error for the new release. If the average effort had been 2 days per error including analysis, correction and retest, then the average effort per error correction in the new release would be projected as 2,5 person days.

In the final step, the number of predicted errors is multiplied by the average effort per error to give the total error correction effort. In our example of 33 predicted errors and an average correction effort of 2,5 person we should budget at least 82.5 person days for error correction. With a release interval of four calendar months this would mean that one person should be assigned the full time task of error correction.

Predicting Adaptation Costs

The costs of implementing change requests are calculated in a similar fashion as the error correction costs, only here other parameters are used. The parameters for estimating adaptation costs are:

- Number of change requests in the last release
- Change request reduction rate
- Average effort per change in the last release
- Adjusted size of the last release
- Adjusted size of the next release

The key questions are what will be the number of change requests in the next release and what will be the average effort required to implement them. This will give the effort required for system adaptation. The number of change requests expected is projected from the number of change requests for the past release adjusted by the change reduction rate. The effort required to correct them is projected from the previous effort per change request adjusted by the difference in the size, complexity and quality of the last and next releases.

The change reduction rate is computed as the average difference between the number of changes per release since the first release. It may indicate an increasing or decreasing growth rate. Empirical studies have shown that the number of change requests per release first goes up and later decreases as the software ripens. After the third or fourth release it may start decreasing at a rate of 10% per release. Thus, if there were 40 change requests for

the last release, then 36 can be expected for the next one.

The average effort per change request in the last release is taken from the time accounting system. From the empirical data collected on the GEOS project it may range between 2 and 4 person days. Assuming an average effort of 3 person days per change request in the previous release and a difference of 1.26 between the adjusted size of the past release and the new release, this gives a projected average effort of 3.8 person days per change request in the new release.

It only remains to multiply the projected number of change requests with the adjusted average effort. In our example, this would be 36 x 3.8 = 137 person days for the implementation of change requests in the next release interval. Together with the 83 person days for error correction, this would amount to 220 person days of effort for the maintenance of the next release.

Predicting Enhancement Costs

Software evolution, as defined here, consists of two parts – functional enhancement and technical improvement. Each has to be treated separately.
The costs of functional enhancement are estimated much in the same way as the costs of a new development. One analyzes the requirements and converts them into a size metric such as function-points or object-points. This size measure is then adjusted by the existing quality and complexity of the system and divided by the productivity rate. It can also be adjusted by influence factors as is the case with the function-point method.

$$Effort = \frac{(Function\_Points \times (Complexity/Quality) \times Influence\_Factor}{Scaled\_Productivity}$$

In the COCOMO-II method, the adjusted size of the new functionality is placed into an equation with a scaling factor of 0,92 to 1,23 where it is multiplied by the product of the influence factors and a system type multiplication factor:

*Effort = SystemType * [(Size/Productivity)**Sf] * InfluenceFactor*

Where SystemType = 0,5:4
Scaling Factor = 0,92:1.23
Influencefactor = Product(Influences(16))

There are other methods as well, such as the object-point method proposed by the author for deriving size from the object model. [30] The important point is to use different methods to size the extent of the functional enhancement and to convert the various adjusted sizes into effort based on the existing productivity rates per size unit, regardless of what that may be.

Predicting Improvement Costs

Improvement is defined here as all activities directed toward the technical perfection of the system including optimization and renovation tasks. If such activities are limited in scope, they will be included within the normal release project. If they are more extensive, they will be assigned to a separately funded reengineering project. In either case, they have to be estimated using a different approach than with functional enhancement.

The big difference is that the size of a functional enhancement, like development, is derived from the requirements, while the size of a technical improvement is derived from the existing code. To estimate a reengineering project, it is necessary to first identify all of those source members, which have to altered, including the components, the database schemas and the user interface definitions, e.g. the HTML and XLS scripts. The size of each member should be adjusted by its complexity and quality. The total size is the sum of the adjusted source member sizes. The size measure here should be either the number of logical statements and declarations, or the uncommented lines of code.

Once the adjusted size of the affected source is known, it can be placed into a COCOMO-II equation to compute the total effort based on past reengineering productivity. Abstract requirement based size measures such as function-points and feature points are equally invalid for reengineering as for maintenance tasks. Reengineering is applied to the code and that presupposes code size metrics. In the case of a redesign it may be more appropriate to use a design size metric like object-points, which counts weighted classes, methods, attributes, interfaces, inheritances and associations to reflect the size of the object model.[32]

The reengineering productivity rate should be based on the past performance of each individual organization, since productivity is known to vary greatly from organization to organization, especially in the field of reengineering where special knowledge and tools are required. The tooling has a much greater effect on reengineering productivity than on development productivity. [33]

Experience with the MainCost Model

For over five years the methods described here were used to project both maintenance and evolution costs in the GEOS product life cycle. In the beginning there was a release every six months. After two years the release interval was reduced to three months. This resulted in 16 releases in all. In the first four releases the prediction of maintenance costs was off by 25 to 40%, mainly due to the underestimation of the number of errors and change requests. After the first four years the maintenance prediction became more accurate as the error rate of circa 350 defects per release persisted from release to release. During the last three years the error rate remained at exactly 0,18 errors per person day. The number of change requests also became more predictable, climbing linearly from 156 in the first release to 281 in the last release.

Another constant factor favoring predictability was the effort per error correction. In the five year period it rose only slightly from 2.6 person days per error in the first year to 3.5 days per error in the last year. The average effort to implement change requests also increased steadily from 3.4 person days in the first year to 4.2 person days in the last year. In the last year the company was investing 4900 person days or 24.5 person years for error correction and 4704 person days or 23.5 person years for adaptive maintenance. These 48 person years amounted to 26% of the total personnel costs.

Throughout the 5 year period under observation some 85 person years were invested in reengineering the GEOS system to reduce complexity and improve adaptability. This amounted to 9% of the total personnel costs. The first renovation projects were not estimated well. There was a variance in the deviation between planned and actual costs of up to 40%. Once the productivity was known, the reengineering estimations became increasingly accurate. A later project to internationalize the system by making it multilingual was overestimated by 3%. This goes to show that once the tools, the methods and the productivity is known, reengineering estimates tend to become highly reliable.

As it turned out, the costs of functional enhancement were the most erratic. As additional customers were acquired, the demand for more and different functionality increased causing the system to double in size from 92.000 function-points in the first year to 185.000 in the fifth year. This increase in system size of 101% required an effort of 6435 person months. This amounted to a median productivity of 14.5 function-points per person month over the 5 years period.

A problem arose from the fact that productivity varied greatly from project to project. While some projects exceeded the median productivity by 70% reaching productivity rates of 24 function-points per person month, other projects reached only half of the median productivity, attaining only 7 function-points per month. There appeared to be different causes of this difference. One cause was weak team leadership, another cause was lack of team experience and a third cause was the project distribution. External teams not on site in Vienna tended to have a much lower productivity ranging from 8 to 16 function-points per person-month.

In one critical enhancement project – the development of an interface to the Swiss stock exchange – outsourced to a daughter company in another city, there was a cost overrun of more than 100% despite the fact that the size was accurately predicted. It was estimated to require 485 function-points and in the end it had 472. The estimate of 24 person months was based on a productivity of 20 function-points per person month. Until the subsystem was operational 52 person months were required giving a productivity of 9 function-points per person month.

The lesson learned is that productivity is the key to cost estimation and that it varies significantly between individuals and organizations. If the person or the organization assigned the task is not known, then the risks of misestimating are high, especially in the case of further development. No matter how accurate the size calculation is, if the productivity projection is wrong the estimate will be off. [34]

Summary and Further Research

In this paper a cost model for software maintenance and evolution has been presented. The model itself evolved out of practical experience with the maintenance and evolution of a large commercial software product. The key to projecting maintenance and evolution costs is to distinguish between various life cycle activities.[35] Trouble shooting has different cost drivers than change management and both differ from functional enhancement and renovation. All life cycle activities have their own particular parameters. What unites them is a common code base from which the planned activities can be sized. What separates them, are different influence factors and their productivity. [36]

There is a definite need for further work in this field. Costs which occur after a system has gone into production need to become predictable. If not, user organizations will never be able to control their costs. This will force them into outsourcing or into using standard off the shelf products. Unfortunately that only passes the problem to someone else. The research community must come to grips with life cycle cost management and provide well founded models based on empirical studies of real world maintenance operations. Only by building up a statistical database will that be possible. Therefore, the first step is to establish corporate or even national metric databases from which researchers can obtain the data they need to create their prediction models. To this end there must be a joint venture on the part of business and government to support such research.

References:

[1] Boehm, B. : Software Cost Estimation with COCOMO-II, Prentice-Hall, Englewoord Cliffs, N.J., 2000, p. 36

[2] Abran, A./ Silva, I./ Primera, L.: "Field studies using functional size measurement in building estimation models for software maintenance", Journal of Software Maintenance, Vol. 14, No. 1, Jan. 2002, p. 31

[3] Sneed, H.: "Estimating the Costs of Software Maintenance Tasks", Proc. of Int. Conf. on Software Mainenance, IEEE Computer Societzy Press, Opio, France, Oct. 1995, p. 168

[4] Putnam, L./ Myers, W. : Measures for Excellence, Prentice-Hall, Englewood Cliffs, 1992, p. 28

[5] Reifer,D./Boehm,B./Basili,V./Clark,B.: "Main-taining COTS Systems – 8 Lessons learned", IEEE Software, Sept. 2003, p. 69

[6] Sneed, H.: "Measuring the Performance of Software Maintenance Departments", Proc. of 1st European Conf. on Software Maint. And reengineering, IEEE Computer Society Press, Berlin, March, 1997, p. 119

[7] Sanker, R., Datar, S., Kemerer, C.: "Software Complexity and Maintenance Costs", Comm. of ACM, Vol. 36, No. 11, Nov. 1993, p. 81-94

[8] Lanning, D./Khoshgoftar, T.: "Modeling the Relationship between Source Code Complexity and Maintenance Difficulty, IEEE Comp., Sept. 1994, p. 35

[9] Bril,R./Feijs,M./Glas,A./Krikhar,L./Winter,M.: "Maintaining a legacy-towards support at the architectural level", Journal of Software Maintenance, Vol. 12, No. 3, May, 2000, p. 143

[10] Kemerer, C., Slaughter, S.: "An Empirical Approach to studying Software Evolution", IEEE Trans. on S.E., Vol. 25, No. 4, July 1999, p. 493-508

[11] Chapin,N./Hale,J./Kahn,K.Ramil,J./Tan,W.G.: "Types of software evolution and software maintenance", Journal of Software Maintenance and Evolution, Vol. 13, No. 1, Jan. 2001, p. 3

[12] Martin, R.J., Osborne, W.: "Guidance of Software Maintenance", U.S. Nat. Bureau of Standards, NBS Pub. 500-129, Dec. 1983

[13] Lientz, B./Swanson,B.: Software Maintenance Mgt., Addison-Wesley, Reading, MA., 1980, p.67

[14] Bersoff,E./Davis,A.: "Impacts of Life Cycle Models on Software", Comm. of ACM, Vol. 34, No. 8, August, 1991, p. 104

[15] Boehm, B.: "A Spiral Model of Software Development and Enhancement", IEEE Computer, May, 1988, p. 61

[16] Nosek, J./Palvia,P.: "Software Maintenance Management – Changes in the last Decade", Journal of Software Maintenance, Vol. 2, No. 3, Sept. 1990, p. 157

[17] van Zuylen, Ed.: The REDO Compendium, John Wiley & Sons, Chichester, 1993, p. 9

[18] Lehman, M., Belady, B.: "Program Evolution, Academic Press, London, 1985, p. 29

[19] Gilb, T.: Principles of Software Eng. Mgt, Addison-Wesley, Wokingham, 1988, p. p. 83

[20] Belady, L./Lehman, M.: "A model of large program development", IBM Systems Journal, Vol. 15, No. 3, 1976, p. 225

[21] Poole, H., Huisman, J.: "Using Extreme Programming in a Maintenance Environment", IEEE Software, Dec. 2001, p. 42-50

[22] Bennett, K., Rajlich, V.: "Software Main-tenance and Evolution – A Staged Model" Future of Software Eng., ICSE-2000, IEEE Press, Limerick, 2001, p. 73-89

[23] Sneed, H.: "Estimating the Costs of Software Maintenance Tasks", Proc. of Int. Conf. on Software Maint., IEEE Computer Society Press, Opio, France, Oct. 1995, p. 168-181

[24] Broessler,P./Sneed,H.: "Critical Success Factors in Software Maintenance", Proc. of ICSM-2003, IEEE Computer Society Press, Amsterdam, Sept. 2003, p. 190

[25] Harrison,M./Walton, G.: "Identifying high maintenance legacy software", Journal of Software Maintenance, Vol. 14, No. 6, Nov. 2002, p. 429

[26] Selic, B.: "The Pragmatics of Model-Driven Development", IEEE Software, Sept. 2003, p. 19

[27] Dromey, D.: "A Model for evaluating Software Product Quality", IEEE Trans. on S.E., Vol. 21, No. 2, Feb., 1995, p. 146-152

[28] Blaha, M.: "A Copper Bullet for Software Quality Improvement", IEEE Computer, Feb. 2004, p. 21

[29] Sneed, H. : "Test Case Analysis", Proc. of CSMR-2004, IEEE Comp. Society Press, Tampere, March, 2004

[30] Maare,M./Bertolio,A.: "Using Spanning Sets for Test Coverage Measurement", IEEE Trans. on S.E., Vol. 29, No.11, Nov. 2003, p. 974

[31] Kajko-Mattson, M., Forsannander, S., Andersson, G.: "Software Problem Reporting and Resolution Process at ABB", Journal of Software Maint.", Vol. 12, No. 5, Oct. 2000, P. 255-286

[32]Hughes, B.: Practical software measurement, McGraw-Hill, London, 2000, p. 143

[33] Sneed, H.: "The Economics of Software Reengineering", Journal of Software Maint., Vol. 3, No. 3, Sept., 1991, p. 163-182

[34] Jörgensen, M., Sjöberg, D.: "Impact of Experience on Maintenance Skills", Journal of Software Maint., Vol. 14, No. 2, April, 2002, p. 123-146

[35] Mens,T./Wermelinger,M.: "Separation of concerns for software evolution", Journal of Software Maintenance and Evolution, Vol. 14, No. 5, Sept. 2002, p. 311

[36] Rajlich, V., Wilde, N., Page, H. : "Software Cultures and Evolution", IEEE Computer, Sept. 2001, p. 24-28