Critical Success Factors in Software Maintenance
A Case Study

Harry M. Sneed & Peter Brössler
Software Data Service, Vienna, Austria
Email: Peter.Broessler@SDS.at, Harry.Sneed@SDS.at

Abstract: *The objective of this paper is to identify those factors, which are critical to the success of a maintenance operation in general and to apply them to a particular maintenance project. The project in question is the maintenance and evolution of a very large and complex banking application system for securities processing which has been in progress since several years. Eight factors are defined and evaluated in accordance with the existing literature on software maintenance and with the experience gained on several such maintenance projects. Each of the eight factors is scored according to a given metric scale. It is left to the evaluator to weigh and evaluate the significance of the individual factors. The study is based on empirical data collected over the duration of the project and is intended to contribute to the overall knowledge of software maintenance management.*

Keywords: *Maintenance and Evolution, Software Product Management, System Measurement and Evaluation, Critical Success Factors, Empirical Studies.*

1. Introduction

The purpose of this paper is to investigate the key factors, which contribute to the success of a software maintenance operation. After having defined what is meant by software maintenance, the paper goes on to address the question – what is success in software maintenance. To this end, it proposes eight potential success factors. For each of these eight factors, it presents metrics, which can be used to measure their degree of fulfillment. Having set up the measurement framework, the paper then presents a study of how this framework has been applied to an ongoing maintenance operation involving a standard banking application. It ends with an explanation of what conclusions can be drawn from the study in regards to success in software maintenance.

2. Subject of the Study

The project described in this paper was started in 1992 and has been going on for more than a decade. The product under development is a large-scale bank application for security trade processing. It not only handles the buying and selling of securities, it also manages the related clearing and settlement procedures and provides a wealth of information to the user and to neighbor systems within the user organization. It is designed to meet the requirements of both very large and small financial institutions. To this end the application has been highly parameterized so that it can be easily adapted to differing requirements. Furthermore it is multilingual, so that it can be localized to any country, including countries where different languages can be used simultaneously. The application can also be used for internationally operating transaction- and service banks, ensuring the proper boundaries between separate entities on one hand and the common processing strategies for regional banking institutions on the other.

At the technical level the product has a classical three level client/server architecture with a PC-Windows client, a Unix or Linux application server and a mainframe data server, which can run either under MVS or Linux. The data entry, checking, and data presentation is made via a windows graphical interface. These front-end functions are implemented in C++ with fat client components. The backend functions for data transmission, data processing and data access are implemented in a macro C language. The data accesses are realized with embedded SQL. The data is stored in a relational database, which may be either DB2, Oracle or SQL Server. The processing of a large number of parallel user transactions is supported by CICS or other transaction monitors. The communication between clients and servers is a proprietary solution based on remote procedure calls. This is part of the custom-made application framework, which acts as the technical foundation for the system as a whole. It handles not only the interactions between clients and servers, but also the connections between the system and the environment and the system and other systems. In this way the application is isolated from its environment and can be readily ported from one environment to another. The technology used is typical of the client/server technology of the 1990's [1].

3. Key Questions

The key questions to be answered here in this paper are what is success in software maintenance and what factors have the greatest influence upon that success. The answer to the first question is a definite prerequisite for the second. Yet there have been many answers to the second

question published without dealing with the first question. As a rule, previous answers have had a narrow focus, most of them dealing with the product, e.g. [2,3,4,5], some with the process, e.g. [6,7,8,9], some with the environment, e.g. [10,11,12] and a few with the human resource factor, e.g. [13,14]. Hardly any have attempted to answer the first question with a few notable exceptions [15,16]. It would appear that success in maintenance is assumed to be obvious. However, it is no more obvious than the term maintenance itself, which has always been a controversial subject.

3.1 What is Software Maintenance?

Even today there is no real agreement on what software maintenance means. As early as 1983 the U.S. General Accounting Office defined software maintenance as being all work done on a software product after it has gone into production [17] It includes corrections, changes, enhancements and optimizations. In the case of enhancements, there has to be a limit to the rate of enhancement to distinguish it from a new development reusing portions of the existing system. Most company policies would agree that the annual growth rate should not exceed 25% to still consider the product to be in maintenance. If it exceeds this limit, then the product is on the verge of becoming another product and the project should be considered a development project. Therefore, enhancement in terms of functional growth, has to have an upper bound.

Adaptive maintenance changes functionality to meet another requirement in place of an existing requirement. Functional enhancement adds additional functionality to meet new requirements. Corrective maintenance only serves to place the software in a state that it should have been in, in the first place, provided that state has been defined.

Perfective maintenance was used by Lienz and Swanson to denote enhancement and optimization [18]. Here it is limited to those measures taken to fulfill nonfunctional requirements including optimization and reengineering. Enhancement increases the value of the product whereas with corrective and adaptive maintenance the value of the product remains constant. Perfective maintenance may increase the value of the software, but not necessarily the value of the application. From an economic point of view this is an important distinction since a value increase to the functionality should also lead to an increase in the price, whereas an increase in maintainability or performance will probably have no effect on the income, it will have an effect on costs.

Of course, getting customers or users to understand these subtle differences is no easy task. That is why it is all the more important to reach an agreement with the users as to what maintenance is and what a maintenance contract should include or not include. It has been recommended that customers should pay for everything other than corrective maintenance to keep the costs of the maintenance operation from running out of control, however in the past it has been difficult to get users to finance perfective maintenance tasks [19]. In the case of standard software products, the users also expect the supplier to finance all adaptive maintenance of a general nature such as changes to the tax laws. So in the end, maintenance has to be defined within each particular context.

In their stages model Rajlich and Bennett distinguish between development, evolution and maintenance phases [20]. The maintenance phase begins when the significant growth of a system stops. The development phase lasts until a system goes into production. In between, where the system is in production but still growing, is the evolution phase.

Belady and Lehman distinguish between p-type, s-type and e-type systems. For p-type systems, there is no maintenance, since these are one time throw away products [21]. For s-type systems, the growth rate is less than 10% per annum, which is more or less equivalent to Rajlich and Bennett's maintenance phase. E-type systems have a growth rate of more than 10% per annum and are considered to be evolving or dynamic systems.

In the case of the GEOS system, it first went into production in 1998, the conceptual development started 1992. In 1998 GEOS had reached a total of 92,000 Function-Points. In the following two years the size increased by two thirds, going up to 154,000 Function-Points. In the last two years the size of the system has only increased by an additional 32,000 Function-Points or by 10 % per annum. Thus, by the classical definitions of the U.S. GAO and the model of Lientz and Swanson, the GEOS project has left development and gone into maintenance, albeit with a dynamic rate of growth. According to the model of Bennett and Rajlich , GEOS is now in the evolutionary stage. In terms of the evolution model of Belady and Lehman, GEOS is very much an e-type system with all of the difficulties related to such systems in that borderland between systems under development and systems in maintenance. It is very close to being what Basili terms as reuse oriented software development [22].

3.2 What is Success in Software Maintenance?

In coping with critical success factors in software maintenance, one has to define precisely what success is within the context of a maintenance project. There is very little in the literature to build upon here. Most previous papers, which have dealt with this topic put forward the simple contention that success in maintenance is to increase user satisfaction while reducing maintenance costs [23]. This may be a good starting point, but it is an over simplification of a very complex issue. This paper is

proposing a multifactor model of measuring software maintenance success in terms of quantifiable objectives. This is in tune with the Goal, Question, Metric Model of Basili, Rombach and associates [24].

The eight success factors proposed here are derived from a maintenance assessment model presented by one of the authors at the ICSM-1996 [25]. This paper both widens and deepens that original model by adding and detailing the following assessment factors:

1. Functionality: The maintenance operation should at least preserve if not enhance the functionality of the system under maintenance.
2. Quality: The maintenance operation should preserve if not increase the quality of the system under maintenance.
3. Complexity: The maintenance operation should not increase the complexity of the product relative to the size.
4. Volatility: The maintenance operation should not lead to an increase in the volatility of the product.
5. Costs: The relative costs per maintenance task should not increase, provided the tasks are of similar scope.
6. Release deadlines: The agreed upon release deadlines should be kept and delays should not increase.
7. User satisfaction: The user satisfaction rate should remain at least at the same level, if not increase.
8. Profitability: Last but not least, the maintenance operation should be profitable or at least cover it's costs.

3.2.1 First Objective: Preserving Functionality

Boehm has pointed out that the first objective of any maintenance manager is the continuity of the service [26]. A user should never have less functionality today than he had yesterday, meaning that every new release must contain as a minimum those functions and data contained by the last release, as long as they are required. A function or a data element may only be deleted if all users agree and there are no undesired side effects. A loss of functionality or information is referred to as functional or informational erosion.

There are two ways of measuring this. The one way is by regression testing. All functional test cases, which were performed for the last release, must also pass through the next release and their results compared. The other method is static source and test case comparison. The source of the last release should match the source of the new release with the exception of added or purposely deleted lines. The same applies to the test cases. The comparison of source and test case deltas together with dynamic trace paths and data results together form a fairly reliable indicator of the functional change rate.

3.2.2 Second Objective: Preserving Quality

An important goal of any software maintenance project is to increase the product quality by the preemptive elimination of error causes, by tuning the system to use fewer resources or by improving the quality of the code and the documentation. However, one needs to be cautious here, as costs can easily get out of hand. Users contract to buy or lease a product as it stands, warts and all. The overall quality of a software product is determined in the development phase. Once the product goes into maintenance, it is both costly and risky to try and raise the level of quality significantly without an explicit contract from the users to do so. This does not, however, preclude local improvements.

It is the responsibility of the maintenance organization to preserve the quality they started with. Quality erosion may be measured in terms of increasing error rates, loss of performance, decreased productivity, increasing inconsistency between documentation and code, and a reduction in code quality. All of these factors can be measured. The error rate is measured as the number of errors relative to the size of the system – error density. Productivity is measured in person-days per impacted size unit. Performance is measured in terms of average response time for online transactions and average execution time for batch transactions relative to the system functionality expressed here in function-points. Of course, if inputs, outputs and accesses are added the performance will drop, but the number of function-points will increase proportionately.

The consistency of code and specifications can be measured by means of static analysis, comparing the entities and relations of the one with those of the other. The consistency of code and user documentation is measured by deriving test cases from the documentation to test the system and then, comparing the actual system behavior with the behavior prescribed in the documentation. This is taken from the literature on software verification [27].

A change in code quality is detectable by means of code auditing. The maintenance phase inherits the code and the coding standards from the development phase. If there was never any standard to begin with it is difficult to install a new standard after the fact. The standard should be a compromise between the state of the code as it is and the way the code should be in the future. As the code improves, the standards can be raised. Thus, the maintenance phase should not only ensure that the code does not get worse, meaning the number of standard violations relative to the number of statements should not increase, but it should also work toward incrementally improving the code while at the same time introducing new standards. The rate of conformity to the standards is a good indicator of quality erosion [28].

3.2.3 Third Objective: Controlling Complexity

Just as the maintenance operation strives to preserve functionality and quality on the one hand, it should strive to control complexity on the other.

In software systems there are two levels of complexity – macro complexity and micro complexity. Macro complexity is the complexity of the system architecture and can be measured in terms of dependencies among architectural layers as well as between components within the architectural layers. It can be measured by means of the classical coupling metric. [29]

An impact analysis will indicate from which other components a given component is dependent upon, i.e. it invokes one of their functions, inherits from one of their classes, includes one of their members, and so on. Each component has such a dependency rate, which is the ratio of the number of components it is dependent on relative to the number of all components in the subsystem. The same metric can be applied to subsystems. A subsystem can be dependent upon other subsystems of a system. The number of these dependencies relative to the number of subsystems is it's dependency rate.

Micro complexity is a question of the internal structure of the components for which there are hundreds of different metrics [30]. Based on whatever metrics one deems appropriate, they should be normalized to a common scale and come up with an average complexity per component. This average complexity should not be increased by the maintenance operation.

Thus, a large system will have different complexities at different levels – at the component level, at the subsystem level and at the system level. The maintenance operation may lead to an increase in the number of components and subsystems, but it should not lead to an increase in their interactions. They should remain at least at the same level, if not decrease as a result of reengineering efforts.

3.2.4 Fourth Objective: Avoiding increasing Volatility

System volatility has been defined as the propensity of an information system to change it's state from evolution to revolution [31]. Revolution demands a new system development life cycle with all of it's accompanying costs and risks. A steady evolution of the product is the goal of the maintenance operation.

System volatility is measured through the volatility index proposed by Heales [32]. This index is determined by the number and the extend of the change requests requiring enhancement relative to the other maintenance tasks. Enhancement is known to cause deep structural change. It should slowly decrease as the system ripens. If it goes up again as is the case with the bath tub curve, this is a sign that the system no longer meets user requirements and needs to be totally revised.

A maintenance operation should ensure that the volatility index decreases or remains at a constant level. A sharply rising volatility index is the sign of extremely high risks.

3.2.5 Fifth Objective: Controlling Costs

The average costs of fulfilling a change request relative to the size of the impact domain is another important criteria for the success of a maintenance project. The impact domain of a change can be defined in terms of function-points, object-points, statements, or any other size measure [33]. The effort in person days relative to the size metric gives the productivity rate. The productivity rate in maintenance will of course never match that of development. Therefore, there has to be a separate maintenance productivity rate based upon the size of the impacted domain and the complexity and quality of the software. This scale is set up at the beginning of the evolution phase and should be monitored annually to ensure that it does not significantly decrease. A decreasing productivity rate usually goes hand in hand with decreasing quality and increasing complexity [34].

3.2.6 Sixth Objective: Keeping Regular Releases

As time passes, the intervals between releases are expected to increase. However they should not deviate from the agreed upon intervals and deadlines. The length of deviations from release deadlines in calendar days is a good metric for evaluating the punctuality of the maintenance operation. Normally the release intervals are part of the service level agreement with the users. They may be set to one year, six months or three months. Whatever they are set to, they should be kept. Deviating from the contracted release interval time is surely a sign of service degradation. Therefore, sustaining the release intervals is a major objective of any maintenance project.

3.2.7 Seventh Objective: Sustaining User Satisfaction

Measuring the degree of user satisfaction is not a simple subject to deal with. It is closer to social science than to computer science and requires a significant investment in time and resources. Any measure of user satisfaction must be founded on a comparison of what the customer feels should be offered and what is ultimately delivered. In addition, the polling needs to be repeated at regular intervals. Questions to be put to the user of a software product may include such criteria as

- satisfaction with the system functionality
- satisfaction with the system quality
- satisfaction with the system performance
- satisfaction with the user support
- satisfaction with the maintenance service

One method recommended in the literature and practiced in the IT service area is the SERVQUAL method of assessing both service expectations and perceptions of deliverables [35]. The SERVQUAL method has two

parts. The first part consists of 22 questions for measuring expectations. These statements are formulated to reveal the degree of service expected by the user. The second part contains the same questions, but phrased to measure perceptions of the actual service delivered.

The degree of user satisfaction is captured by a gap score (G) indicating perceived quality of a given service where G=P-E, whereby P is the sum of the delivered services and E is the sum of the expected services. A positive score shows that the user is getting more than what he or she expects. A negative score indicates that the user is getting less than what he or she expects. This polling of the users should take place at least once a year [36].

For a maintenance operation to be considered successful, the degree of user satisfaction, or gap score, should be increasing or, at least, remaining at the same level. Decreasing user satisfaction is one sign of service degradation. However, it is not the only one. There is also the degree to which the IT system contributes to the profitability and to the competitiveness of the user organization. This has to be measured by a reoccurring cost/benefit analysis. A successful maintenance operation will be satisfying the end users while at the same contributing to the financial well being of the user organization.

3.2.8 Final Objective: Covering Costs

The ultimate success goal is a matter for the company accountants. The annual fees for the support and correction of a system plus the additional annual income for changes and enhancements should amount to at least as much as the annual costs of the maintenance operation. Error correction, system upgrades, and customer support may be covered by the standard maintenance fees. Adaptations and enhancements should be charged to the users requesting them. In some cases, such as large scale reengineering projects, it will be necessary to share the costs among all users. The Euro conversion is an example of such a project. A technical conversion to another platform is another example. Customers must be educated to the fact that the slightest change to an existing software system occurs costs, both direct and indirect. Thus, attaining this first goal is primarily a contractual challenge and secondly a challenge for the maintenance task estimators.

4. Rating the GEOS Software Maintenance Operation

Having set up these eight success criteria, it is now the purpose of this contribution to illustrate their application to the maintenance and evolution of the GEOS standard software package. Some might say that fulfilling the last criteria – covering your costs – is all that matters. Others might claim that user satisfaction or the contribution of the software product to the profitability and competitiveness of the user organizations is the primary objective. All of the other criteria – functionality, quality, complexity, volatility, productivity and timeliness – are all related to one of the two KO criteria – either to costs or to user satisfaction. They are included to demonstrate the relation between hard, precisely measurable, factors such as reliability and soft, imprecise measurable, factors such as user satisfaction.

4.1 Rating the Functionality of the GEOS Product

In terms of function-points, the functionality of the GEOS product has increased by 200% from 92.079 when it first went into production at the end of 1998 to 185.830 at the end of 2002. With the exception of the year 2001 when there was a major reorganization the growth rate has been steady as shown below:

Year	Funct-Points	Increase
1998	92,079	100%
1999	108,006	17%
2000	153,830	42%
2001	160,936	5%
2002	185,830	15%

Another way of viewing the functionality of a system is in terms of the use cases, the user interfaces, the reports, the database tables and the data interfaces. At the beginning of 1999 GEOS had :

1855	usecases
735	user interfaces
64	reports
703	data interfaces and
400	database tables

Now it has:

3912	usecases
932	user interfaces
401	reports
2149	data interfaces and
638	database tables

The static analysis of both the specification and the code indicates that functionality has doubled. The regression tests, which are based on the test cases executed in the last release ensure that no functionality has been lost. All of the use cases with all of the user interfaces tested in the last release are also tested in the new release. Thus, one can assume that this criteria has been fulfilled.

4.2 Rating the Quality of the GEOS Product

Product quality is a multiple dimensional factor as has been pointed out in the ISO-9126 standard [37]. There is quality in terms of reliability, performance, usability, portability, maintainability, interoperability, and reusability. For the sake of simplification the internal quality features such as maintainability, portability, interoperability and reusability have been grouped together under the rubric construction. Usability is an issue for the last success criteria – user satisfaction. So

here only reliability, performance and construction are considered.

Reliability is measured in terms of defect rates relative to the size of the system. The GEOS project distinguishes between three error classes – major errors, medium errors and minor errors. Major errors cause the system to fail or cause damage to the user in some way. Medium errors produce wrong results, which can be handled by the users or a system failure, which can be avoided. Minor errors affect only the user's view of the system.

In the first year of production GEOS had some 644 customer errors reported, of which 242 were major errors, 240 medium errors and 162 were minor errors. Given the size at that time of 92,000 Function-Points, this amounted to an error density of 0.007. In the year 2002 1048 customer errors were reported of which 651 were major errors, 268 were medium errors and 129 were minor errors. As shown below, the error rate has increased in absolute terms, but the error density has decreased to 0.006 as the system grew to it's current size.

Year	Errors reported	Size (Fkpt)	Error Density
1999	644	92,079	0.007
2000	872	108,006	0.008
2001	612	153,830	0.004
2002	1048	160,936	0.006

It is obvious that the reliability of the product has not really improved over the years, but neither has it gotten significantly worse. The maintenance operation has succeeded in maintaining the reliability rate despite significant growth.

The performance of GEOS is measured according to three criteria:

- response time
- throughput and
- resource utilization

The response time is the time interval between the receipt of a transaction on the server to the termination of that transaction. Throughput is the number of transactions that can be processed in a given time interval. Resource utilization is the degree to which the system capacity is occupied by the GEOS application. The baseline environment is the IBM Z series with an OS/390 operating system.

As depicted below, the performance figures have actually improved over the years despite the system growth of 100%. When GEOS first went into production, the average transaction time was 41 microseconds with a throughput of 24 transactions per second and the system resource utilization averaged at 48%. Currently the response time is measured at 21 microseconds for 47 transactions per second and the system resource utilization has fallen to 42%. This improvement is of course not only a result of system tuning, but also of the faster machine. Independent of the reason, the system performance has not been negatively influenced by the system evolution.

Year	Transactions	Avg. Response Time
1999	28 per sec	0.036 Secs
2002	47 per sec	0.021 Secs

Year	System Size	System Utilization
1999	1,251,675 Stmts	55 %
2002	2,501,061 Stmts	42 %

The construction quality is the median quality coefficient of the quality characteristics modularity, portability, testability, reusability, flexibility, interoperability, and conformance. These internal quality metrics have been described in previous publications [38]. They are derived from a static analysis of the code repeated every six months. In it's original state, the GEOS code had a median construction quality rate of 0.639. Over the past four years this rate has slightly increased to it's current state of 0.662.

Year	Components	Median Quality
1999	2038	0.639
2000	2703	0.643
2001	3265	0.647
2002	3488	0.662

These code quality metrics indicate that the GEOS maintenance operation has managed to preserve the original construction quality and even to increase it.
So as far as the preservation of quality is concerned, the GEOS maintenance operation can be considered a success in that it has fulfilled all three of the quality rating criteria.

4.3 Rating the Complexity of the GEOS Product

GEOS product complexity is measured at both the macro and the micro level. At the macro level it is the number of interactions, i.e. data flows and function calls, between subsystems relative to the number of subsystems. At the micro level it is the median complexity rate of eighth different complexity metrics – Chapin's Data Q-complexity, Dataflow complexity, Cards's Data Access complexity, Henry's Interface complexity, McCabe's Cyclomatic complexity, McClures Decisional complexity, Sneed's Branching complexity and Halstead's language complexity. The use of these complexity metrics has been documented in previous publications [39].

The average micro complexity rate of all components has been decreased over the four year period since the beginning of 1999 from 0.622 to the current rate of 0.590. It might be expected that the complexity of the code would rise as supposed by Belady and Lehman on their study of the IBM-TSO system, but in the case of GEOS the complexity of the code has actually decreased as a result of clean-up operations.

Year	Components	Median Complexity
1999	2038	0.622
2000	2703	0.633
2001	3265	0.582
2002	3488	0.590

The average macro complexity, or coupling rate, for all subsystems has risen from only 0.250 in 1999 when there were only six subsystems – three frontends and three backends – with 8 interactions to 0.482 in 2002 where there are 29 subsystems – 12 frontends and 17 backends.- with 56 interactions. This increasing global complexity is due in the most part to the continual reorganization of the system to meet new requirements. It is a certainly a matter of concern, but it also typical of a rapidly evolving system. In summary it can be said that although complexity is being reduced at the micro level, it continues to grow at the macro level.

4.4 Rating the Volatility of the GEOS Product

Product volatility is measured in terms of the effort going in to enhancing the product. It is expected to be high at the beginning of the evolution phase as missing requirements are added, but to decrease over time as the system slowly covers the application at hand until it levels off at a flat rate of growth. This bath tub curve may hold true for well defined back office applications, but not for such dynamic front office applications like stock trading. Here the software system is trying to ride a tiger.
.
GEOS is a good example of a highly volatile system. In absolute terms, the number of change requests has increased threefold since 1999.

Year	CRs
1999	568
2000	703
2001	1159
2002	1802

The amount of effort for enhancing the system relative to the effort for changing and correcting GEOS has remained relative constant over the past four years. In 1999 16,005 person days were booked against further development as opposed to 10,121 for maintenance. In 2002 it was 12,464 person days for further development and 15,321 days for maintenance. This indicates that the product is a long way from satisfying all user requirements and in covering the application area. Thus the volatility rate remains high.

Year	Maintenance	Enhancement	Volatility
1999	10121 PDs	16005 PDs	0.612
2000	16519 PDs	15810 PDs	0.489
2001	17528 PDs	14332 PDs	0.449
2002	15321 PDs	13464 PDs	0.467

This high volatility rate can not be attributed to the maintenance operation as such. It is not so much a sign that the product has not been built right, but more a sign that the right product has yet to be built. Nevertheless, it does have an effect on the costs of the maintenance operation and should be included to round out the definition of success. Maintaining and developing a product at the same time is a challenging task which has to be very well managed if it is to succeed.

Thus, as far as volatility is concerned, GEOS can not be considered a success. The rate of expansion is far more than it should be if the right product had been built from the beginning. The constant adding of new functionality makes it difficult to stabilize the system and adds to the increasing entropy.

4.5 Rating the Productivity of the GEOS Maintenance Operation

Productivity is measured by taking a given size measurement such as code lines, statements, function-points or object-points and then dividing it by the number of person days worked on that specific project.

Size-Metric / Effort

In the case of maintenance, this is not so easy since one is often only changing an existing component. In this case one must adjust the size of the component by the change rate. If a given component has 50 function-points and the code is changed by 10%, i.e. 10% of the code lines are deleted, overwritten or added, then the change amounts to 5 function-points. If six person days were required to perform the task, the productivity is 0.83 function-points per person day.

Previous maintenance studies by Vessey and others have demonstrated how productivity sinks as complexity increases and quality decreases [40]. In the case of GEOS code complexity has actually decreased and code quality increased over time. In comparison, productivity has remained constant. In 1999 the GEOS developers were performing at an average rate of 0.72 function-points or 8 statements per person day. In 2002 the productivity had decreased slightly in function-points per day to 0.69, but it increased to 8.6 statements per day. Considering the fact that productivity can not be measured precisely, especially not in maintenance, the conclusion is that productivity has remained fairly constant as shown in the following extract from the GEOS maintenance productivity table:

Year	FP's per PD	Stmts per PD
1999	0.72	8.2
2000	0.68	8.8
2001	0.67	8.5
2002	0.69	8.6

The conclusion to be drawn here is that contrary to the popular assumption that productivity sinks as software products age, it is possible to sustain productivity in maintenance by constantly improving the quality of the code and documentation as well as the quality of the process.

4.6 Rating the Punctuality of the GEOS Releases

After going into production in 1998 GEOS had a release cycle of every six weeks with an average delay of 2-5 days. At that time at most 15% of the components were affected by the release. As the system has matured, the release intervals have been prolonged to every 3 months with no delay at all although now up to 50% of all components are changed in a release. Thus, punctuality has actually increased over the past four years, due to the rigid release schedule.

4.7 Rating the Satisfaction of the GEOS Users

The SDS is now planning to conduct a user survey based on the SERVQUAL method described above. Some of the questions will be deleted which are not relevant to the sale and support of a product. Others will be added to solicit the opinion of the users on the quality of the product and the maintenance process. The goal is to have a service-oriented maintenance organization as described by Niessink and van Vliet using the IT service capability maturity model as a guideline [41].

4.8 Rating the Profitability of the GEOS Maintenance Operation

Profitability is determined by subtracting the costs of the maintenance operation from the income obtained through maintenance fees plus charges for enhancements and changes to the existing software product. Income from the sale of a core product is not included since this should be booked against the original development costs. Income for the development of add on systems is also not included, as this goes under the rubric of further development. To assess the profitability of the maintenance operation, accounting must somehow separate maintenance from development costs.

Since first going into operation at the end of 1998 the GEOS product has been maintained at a slight loss, primarily due to the lack of distinction between true maintenance tasks and development tasks brought about by new requirements. Many concessions have been made to current users with the perspective of adding functionality to the product so that it is more attractive to potential users. This strategy works, if the demand for such a product is large enough. However, it should not be carried too far. A risk analysis should accompany all major enhancements and should be supported by current users to avoid evolution of the product from going off the deep end. In any case, one should clearly distinguish between maintenance and further development for the sake of accurate accounting and sharing costs with users.

5. Summary of Critical Success Factors

This paper attempts to define the most critical success factors for a software maintenance operation based on the existing literature and on the experience gained in the case study. Eight factors have been identified and described in terms of the ongoing maintenance operation. The jury is still out on the last two decisive factors – user satisfaction and profitability. However in accordance with the other six factors – preservation of the existing functionality, quality, complexity, volatility, productivity and punctuality – the GEOS maintenance project can be considered to be relatively successful.

We have argued that the quality of an evolving software system will not necessarily decrease while the complexity increases. With GEOS this is only partially true. Inner modular complexity has actually decreased whereas global complexity has gone up. Through continual rework, it has been possible to keep quality at approximately the same level as it was from the beginning. The same applies to productivity and release punctuality, which have remained constant. Also refuted is the claim that the volatility rate of a product under maintenance will necessarily decrease after the first few years of production. This applies only to static systems. Dynamic systems keep growing until they reach some upper bound of complexity or until they have fulfilled all the requirements of all users. GEOS has yet to reach either limit. In this respect there is a very fuzzy line between evolutionary software development and e-type system maintenance. The growth rate of GEOS is definitely too high for a system that old.

An additional factor to be considered is, of course, the technological basis upon which the system was built. GEOS was built as a client/server application. In the meantime, the client/server technology has been superseded by web-based technology. This puts GEOS in the category of an outdated system, which makes it less attractive for potential users. This is not a maintenance problem as such, but it does decrease the value of the system as a whole.

6. Further Research Directions

There is a great need to have measures of success in software maintenance. Without them, we are not able to assess the effect of processes, methods, tools or techniques upon success. Therefore, the first step is to reach a consensus on what the success factors are and how they can be measured. This paper was intended to further that discussion.

The next step will be to survey all of the organizational, procedural and technical attributes, which could relate to

the success factors and to perform a correlation analysis upon them. The goal of that work will be to find which current practices and which potential innovations correlate most with the success factors presented here. This will allow the GEOS management to preserve successful practices and to promote only those innovations, which promise to contribute to the success factors.

References:

[1] Sadiq, W., Cummins, F.: Developing Business Systems with CORBA, Cambridge University Press, Cambridge, 1999, p. 23-42

[2] Grady, R. B: "Measuring and Managing Software Maintenance", IEEE Software, Sept. 1987, p.35-49

[3] Gibson, V., Senn, J.: "System Structure and Software Maintenance Performance", Comm. of ACM, Vol. 32, No. 3, March 1989, p. 347-358

[4] Rombach, D.: "A controlled Experiment on the Impact of Software Structure on Maintainability", IEEE Trans. on S.E., Vol. 13, No. 3, March, 1987, p. 344-361

[5] Briand, L., Bunse, C., Daly, J.: "A controlled Experiment on the Maintainability of Object-Oriented Systems", IEEE Trans. on S.E., Vol. 27, No. 6, June 2001, p. 513-530

[6] Prechelt, L.,Unger, B. ,Tichy, W., Broessler, P., Votta,L.: "A controlled Experiment in Maintenance comparing Design Patterns to Simple Solutions", IEEE trans. on S.E., Vol. 27, No. 12, Dec. 2001, p. 1134-1144

[7] 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

[8] Sherer, S.: "Using Risk Analysis to manage Software Maintenance", Journal of Software Maint.", Vol. 9, No. 6, Dec. 1997, p. 345-264

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

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

[11] Swanson, E.B., Beath, C.: "Department-alization in Software Development and Maintenance", Comm. of ACM, Vol. 33, No. 6, June, 1990, p. 658-667

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

[13] Ramaswamy, R.: "How to staff business critical Maintenance Projects", IEEE Software, June, 2000, p. 90-94

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

[15] O'Neill, D.: "Software Maintenance and Global Competitiveness", Journal of Software Maint., Vol. 9, No. 6, Dec. 1997, p.379-400

[16] Sahin, I., Zahedi, F.: "Policy Analysis for Warranty, Maintenance and Upgrade of Software Systems", Journal of Software Maint., Vol. 13, No. 6, Dec., 2001, p. 469-495

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

[18] Lientz, B., Swanson, E.B.: Software Maintenance Management, Addison-Wesley, Reading, 1980, p. 105

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

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

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

[22] Basili, V.: "Viewing Maintenance as reuse-oriented Software Development", IEEE Software, Jan. 1990, p. 19-25

[23] Guimares, T.: "Managing Application Program Maintenance Expenditures" Comm. of ACM, Vol. 26, No. 10, Oct. 1993, p. 739-746

[24] Basili, V., Caldiera, C., Rombach, H.-D.: "Goal, Question Metric Paradigm", Encylopedia of S.E., Vol. 1, John Wiley & Sons, 1994, p. 528-532

[25] Sneed, H.: "Evaluating the Maintenance Process at the Zurich Life Insurance", proc. of Int. Conf. on Software Maintenance, IEEE Computer Society Press, Monterey, 1996, p. 217-227

[26] Boehm, B. "The Economics of Software Maintenance", in Proc. of Int. Conf. on Software Maint., IEEE Computer Society Press, Monterey, Dec. 1983, p. 9-36

[27] Chechik, M., Gannon, J.: "Automatic Analysis of Consistency between Requirements and Designs", IEEE Trans. on S.E., Vol.27, No. 7, July 2001, p. 651-672

[28] Coleman, D., Lowther, B., Oman, P.: "Using Metrics to evaluate Software System Maintainability", IEEE Computer, August 1994, p. 42-57

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

[30] Harrison,W, Magel, K., Kluczy, R., DeKock, A.:"Applying Software Complexity Metrics to Program Maintenance", IEEE Computer, Sept., 1982, p. 65-79

[31] Truex, D., Baskerville, R., Klein, H.: "Growing Systems in Emergent Organizations", Comm. of ACM, Vol. 42, No. 8, August, 1999, p. 117-129

[32] Heales, J.: "A model of factors affecting an information system's change in state", Journal of Software Maint., Vol. 14, No. 6, Dec. 2002, p. 409-428

[33] 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

[34] Abran, A., Silva, I., Primera, L.: "Field Studies using functional size Measurement in building estimation models for Software Maintenance", Journal of Software Maint., Vol. 14, No. 1, Feb. 2002, p. 31-64

[35] Jiang, J., Klein, G., Tesch, D., Chen, H-G.: "Closing the User and Provider Service Quality Gap", Comm. of ACM, Vol. 46, No. 2, Feb. 2003, p. 72-76

[36] Ferguson, J., Zawacki, R.: "Service Quality – A critical success factor for IS Organizations", Information Strategy, Vol. 9, No. 2, Dec. 1993

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

[38] Sneed, H., Merey, A.: "Automated Software Quality Assurance", IEEE Trans. on S.E., Vol. 11, No. 9, p. 909-916

[39] Sneed, H.: "Understanding Software through Numbers", Journal of Software Maint., Vol. 7, No. 6, Dec. 1995, p. 405-420

[40] Vessey, I., Weber, R.: "Factors affecting Program Repair Maintenance", Comm. of ACM, Vol. 26, No. 2, Feb. 1983, p. 128- 133

[41] Niessink, F., van Vliet, H., "Software Maintenance from a Service Perspective", Journal of Software Maint., Vol. 12, No. 2, April, 2000, p. 103-120.

Proceedings of the International Conference on Software Maintenance (ICSM'03)
1063-6773/03 $17.00 © 2003 IEEE