International Journal of Computers and Application (IJCA)
DOI: 10.2316/Journal.202.2010.2.202-2774

VALUE-DRIVEN SOFTWARE MAINTENANCE

Harry Sneed
ANECON GmbH
Vienna, Austria
harry.sneed@t-online.de

Shihong Huang
Computer Science & Engineering
Florida Atlantic University
shihong@cse.fau.edu

Abstract

This paper extends the concept of value-based software engineering as proposed by Boehm to the field of software maintenance. The various approaches to assessing the value of software products and to calculating the return of investment (ROI) on software projects are reviewed. The authors propose a methodology of value-driven software maintenance for assessing the value of software maintenance based on Hayek's theory of maintaining the value of capital goods in an evolving capital structure. A method for predicting costs of maintenance tasks based on impact analysis is combined with a method for assessing the added value produced by the maintenance tasks to compute the ROI on software projects. The validity of the approach is illustrated through a case study of the German social services payment system.

Keywords: software maintenance, value-driven, return on investment (ROI), maintenance cost estimation, impact analysis

1 Introduction welcome

Value–based software engineering emerged out of a synthesis of software project management and business economics. A project consists of a number of tasks. Each task produces a result, e.g., a document, a design artifact, a code module, a set of test cases, and so on. Each result has a value

relative to its proportion of the sum of all results, namely the software product. By delivering a result, a project element task earns the value assigned to it, hence the term “earned value”. The project state (i.e., the degree to which it is complete) can be measured in terms of the number of values that have been earned up to that point in time, relative to the sum of all values. Boehm used this method to measure progress in software projects at TRW and describes it in his first book on software engineering economics [1]. Earned-value project management systems provide a useful technique for monitoring the progress of complex software projects, but they do not address the value of the system being built from the point of view of the stakeholder. This topic has been neglected until recently when value-based software engineering [2] emerged as a new approach to justifying software projects.

This paper applies value-based software engineering concepts to software maintenance tasks. It proposes a methodology for assessing the value of a software maintenance project based on Hayek’s theory [3][4]of maintaining the value of capital goods in an evolving capital structure. The methodology combines impact analysis with a method for assessing the added value produced by the maintenance tasks to compute the ROI on software projects. The validity of the approach is demonstrated through a case study.

The next section provides an overview of the fundamental technologies underlying value-driven software engineering and software maintenance. Section 3 applies value-driven software engineering to software maintenance tasks. To illustrate the proposed approach, Section 4 presents examples of how to assess the value and the benefits of a software maintenance project, and uses the German Social Aid System as an example to demonstrate the approach in a real-world setting. Section 5 summarizes the value-driven software maintenance issues discussed in this paper, and points to some of the areas for future research.

2 Value-Driven Software Engineering and Software Maintenance

Value–based software engineering as proposed by Boehm [2] combines a number of approaches including participatory design, user engineering, cost estimation, software economics, software investment analysis, and software engineering ethics. It is closely related to software maintenance cost prediction and assessing maintenance benefits. This section first briefly describes the key points of value-driven software engineering, and its impact on software maintenance. Section 2.2 discusses some of the alternative approaches to maintaining software systems.

2.1 Value-driven Software Engineering

The essence of value–based software development is that every task produces some result and that this result is judged on the basis of its return on investment (ROI), which is defined as:

$$ROI = (benefit - cost) / costs$$

In the true capitalist spirit, each software artifact produced, whether it is a design document, a code module or a test case, has a value and a cost. The value should be greater than the cost (which is not always true), and its ROI (the ratio of its value relative to its cost) must be comparable with that of other artifacts. Those tasks with the highest ROI are given priority over those with a lesser or a negative ROI. Thus, a rational selection criterion is provided to assist project managers in deciding what to do next [5].

Little [6] reports on an approach applied at his graphics company to compute the value of a software product dynamically. The value is the amount that users would be willing to pay for a feature if it is delivered immediately. The costs are derived from the size of the feature divided by the current team productivity. Novel in their approach is that they use the effort required to produce a value to obtain the time it would take to deliver that value. As pointed out by Boehm, calendar time as a factor

of effort and project compressibility can be expressed as an equation [7]. The value of a software product or service is time sensitive. It may increase or decrease over time depending on market development. As a rule, it tends to diminish as other alternatives come up. Therefore, time is essential. The longer it takes to produce value, the less it is worth.

In his book "Value-based Software Engineering," Biffl [8] summarizes the key issues in determining the value of a software engineering activity, such as the benefits of the product or service minus the costs of product failure, etc. Since value is relative, the value of each activity must be viewed in terms of its relation to the other activities. Activities that provide more benefit at less cost should be given priority over activities that deliver equal benefits at higher costs [3]. The prerequisite to value–driven software engineering is the ability to quantify benefits and to estimate costs [9].

2.2 Alternate Approaches to Maintaining Software Systems

The notion of value-driven software maintenance as presented here is based on the economic advantage of preserving software as it is and only making incremental changes when necessary. An alternative approach is to view software maintenance as a continual development process with a significant reuse factor. In regular intervals a new release of the software product will be put together consisting of some new components and many old components taken from the last release. In effect, each new release amounts to a new development with a high rate of reuse. Such an approach is appropriate for applications developed to solve transient problems or problems of an evolving nature such as marketing applications in business or software produced for different models of the same vehicle. This corresponds to Lehman's E-Type systems in the laws of software evolution [10]. The advantage of this approach is that it supports continual renewal of the software. However with this reuse approach there are higher costs and higher risks involved.

Another more radical approach to evolution is to simply discard the existing software when it no longer suffices or when it no longer fits the technical environment and to start over from scratch. This is referred to as the throw away approach by Belady and Lehman. This approach ensures that the software is always technically up to date and that the solution fits exactly to the problem at hand. From an economic standpoint it is highly irresponsible. No financial manager in his right mind would subscribe to it. It means throwing away a large investment in money and accumulated knowledge. The sad fact is that it takes a long time for software to ripen, often many years. It is an economic loss to disrupt the ripening process of a solution and to start up a new one just because the technology has changed. The costs of the lost knowledge and lost stability are much higher than the benefits of any new technology.

3 Applying Value–driven Software Engineering to Software Maintenance

Software maintenance, as defined by IEEE Standard 1219, "includes modification of a software product after delivery or to adapt the product to a modified environment" [11]. The literature on software maintenance recognizes four distinct types of maintenance tasks: *corrective maintenance*, *adaptive maintenance*, *perfective maintenance*, and *enhancive maintenance*.

Corrective maintenance is concerned with removing defects. Adaptive maintenance is directed toward changing existing functions (e.g., using a different algorithm to compute an already existing output). Perfective maintenance is devoted to increasing the quality of an existing function. It may become faster, or easier to use, or more convenient to adopt (more changeable). Enhancive maintenance implies adding additional functionality to an existing product (e.g., delivering outputs that did not exist before [12]).

Maintenance tasks are triggered by an error report, a change request, or a new requirement. Each error report, each change request, and each new requirement must be subjected to a cost benefit analysis aimed at calculating the return on investment of that request [13]. The three distinct triggers for maintenance activities are shown in

Table 1:

Table 1: Maintenance Triggers and Activities

Trigger	Maintenance Type
Error Report	Correction
Change Request	Adaptation, Perfection
New Requirement	Enhancement

There are two prerequisites to calculating the ROI of a maintenance task: (1) to predict the costs of that task, and (2) to assess the benefit of that task.

The following two sections describe how these two tasks can be performed in detail.

3.1 Predicting the Costs of Maintenance

Estimating maintenance costs can be done using experience or an automated impact analysis.

Les Hatton [14] published an article that was based on a study of 957 maintenance tasks performed on 13 different software products covering from March of 2001 to November of 2005. 10% of the maintenance tasks were corrective, 10% were adaptive, and 50% were perfective and enhancive. Hatton does not distinguish between these two categories. The purpose of the study was to investigate how well the costs of the maintenance tasks were estimated. Hatton observed that 77% of all effort

was spent on tasks that were estimated correctly and that 16% of the effort was spent on tasks that were grossly underestimated. The remaining7% of the effort was in tasks that were slightly underestimated. Overall, one in four of the maintenance tasks were incorrectly estimated, and most of these were completely wrong so that there was no correlating between the estimated and the actual maintenance effort.

An automated process for calculating the costs of a maintenance task was implemented by Sneed at a Viennese Software House [15]. It begins with the analysis of the change request. The artifacts affected by the changes can be classified as primary impacted classes, secondary impacted classes (i.e., derived classes and dependent classes of the changed class), and tertiary impact classes (i.e., the classes depended upon classes in the secondary impact domain). The total size of the impacted domain is given by:

$$\begin{aligned} \textit{Raw change size} = \quad & \textit{(primary impacted class size * change rate)} \\ & + \textit{(}\Sigma\textit{ secondary impacted class sizes * (change rate/2))} \\ & + \textit{(}\Sigma\textit{ tertiary impacted class sizes * (change rate/4))} \end{aligned}$$

The raw change size is adjusted for both the complexity and the quality of the impact domain. The final adjusted size of the code impact domain is the raw size of that domain adjusted by complexity and quality metrics:

$$\textit{adjusted size = raw–size * (complexity / 0.5) * (0.5 /quality)}$$

The total cost of maintenance can be estimated as the sum of the costs of all the individual maintenance tasks adjusted by some overhead factor. This automated estimation process was validated on over 200 maintenance tasks on a large scale banking system with some 2.5 million C/C++

statements with a standard deviation from the actual effort of +/- 17%. (Note: for a detailed description please see [15]).

3.2 Assessing the Benefits of Maintenance

The one side of the value–driven maintenance equation is the cost of performing the maintenance action. The other side is that of the benefit provided by that maintenance action. Here again one must distinguish between the benefits of corrected errors, adapted functionality, improved quality, and added functionality since each type of maintenance work produces another value.

3.2.1 *The Benefits of Corrective Maintenance*

The benefit of an error correction is the inverse of the error costs to the user. Every hour of work lost as a result of a software error is equivalent to the currency value of that error on the benefit side. There are errors whose costs are obvious, and other errors whose real costs are hidden. The estimator must try to include also those hidden costs. Boehm and Huang suggest using a negative Pareto distribution for computing total value loss [16]. Except for trivial errors, such as misspellings on the user interface, error corrections will always have significant benefits even if one only counts the work hours lost as a result of those errors. These benefits justify the priority given to error correction as opposed to other maintenance activities.

3.2.2 *The Benefits of Adaptive Maintenance*

The benefits of adaptive maintenance can be seen as the difference between the value of a software product without the change and the value of the software with the change. In some cases, such as new laws or currency transitions, software becomes valueless if it does not reflect the changes. The value of the adaptation is equivalent to the value of the entire software. In other cases, change is not absolutely essential, but it may lead to increased productivity or cost reduction. If a change allows end users to

perform their job better, their productivity will go up. Added value is a very important concept for justifying change requests [17].

The benefit of an adaptation can be defined as the added value of the system brought about by the change:

$$Benefit = Valueof\ AdaptedSystem - ValueofOriginalSystem$$

3.2.3 *The Benefits of Perfective Maintenance*

Effective software development depends on effective communication among developers and maintainers [18]; hence it is well worth it to optimize that communication by creating comprehensive code. Perfective maintenance is aimed at making the code more comprehensive. It is also aimed at rendering the code more amenable to change and to lessening the probability of error when changing the code.

Sneed suggests measuring the size, complexity, and static quality of code before a reengineering project and then to measure the same after the reengineering project. The value increase can be seen as the sum of the reduced size plus the reduced complexity plus the quality increase [19]:

$$Value = (OldSize/NewSize - 1) + (OldComplexity/NewComplexity - 1)$$

$$+ (NewQuality/OldQuality - 1)$$

A similar approach to assessing the business value of migrating to Web services has been proposed by Tilley, Huang *et al*. [20]. In their approach, they used a healthcare industry problem to illustrate the complex and multifaceted Web services adoption challenges. The goal here was to preserve the functionality of the healthcare system while at the same time giving the users a modern interface and allowing them to reconstruct their work flows using the individual web services as building blocks. In this respect the approach taken by these authors enhanced the value of the software. Beck and Sneed

are concerned with preserving current value whereas Tilley and Huang are concerned with adding additional value. The added value can be computed in terms of the business value of the modern user interface the higher productivity of users, and the increased flexibility of being able to combine web services in different ways when creating new business processes. This is a typical reuse approach that will be expounded upon later. The point to be made here is that 2/3 of the costs of a web-based business process are due to the development and testing of the underlying services. If these services can be taken from the existing software, the user organization can save perhaps ½ of the costs of setting up a service oriented architecture. Thus, while Beck and Sneed are addressing the saving of maintenance costs, Tilley and Huang are aiming at saving development costs.

4 Assessing the Value of a Software Maintenance Project

The value of a maintenance task must be assessed in terms of the proportion of the capital good to which this task applies. If a change affects 5% of the code where the total code has a value of $1 million, and the cost of that change is 3 man-months = $30,000 then the ROI is:

$$(50.000 - 30.000) / 30.000 = 0.66$$

This model would of course assume that all code is equally important, which is generally not true; some components are more important than others. If the code changed has a weight of 1.5 then the ROI would be:

$$(75.000 - 30.000) / 30.000 = 1.5$$

In this way the changing of strategic components would be given priority over the change of less important ones.

In order to apply this seemingly simple model for calculating the ROI, two prerequisites have to be fulfilled. One is to assess the benefit; the other is to estimate the costs. Both have been the subjects of endless research.

4.1 Assessing the Benefits of Maintenance

In his early work on reengineering economics, Sneed defines the benefit of a good or service as being the monetary difference between having a good or service and not having it [21]:

Benefit = (State with good or Service) – (State without good or Service)

Using this definition, it is even possible to have a negative benefit, namely that it costs the owner of a good more to use it than when he does not use it.

According to Sneed, value is the difference between benefits and costs adjusted by risks [22]:

*Value = Benefit – (Costs * Risks)*

Sneed's value equation has been expanded on in a book on the "Preservation of Software Value" by the SAP economist Eckhart von Hahn [23]. Von Hahn is concerned with the maintenance and enhancement of software products as a capital good in an evolving capital structure. He sees reengineering as a means of restoring value to a product whose value is decreasing over time. Von Hahn points out that the benefit of a software system may remain constant but that the costs of maintaining it increase. Therefore, the value drops. To offset this loss of value, SAP has recently raised its annual maintenance fee from 17% to 22% of the selling price. Doing this raises the benefit of the products to SAP, thus offsetting the loss of value due to increasing maintenance expense, but at a cost to the customer. The customer is left with the same solutions for which he has to pay 5% more.

A good example for the application of the Sneed value theory is a software tool for analyzing requirement texts to extract test cases. This job could also be done manually and usually is. To accomplish this task manually on a web portal for the State of Saxony in Germany, the author required 18 person days at a cost of €800 per day = €14,400. To do the same job with a text analysis tool costs 3 person days. This gives a net savings of 15 person days = €12,000. The costs of maintaining the tool is about €4,000 per month. The risks of using it and having problems with it are 30%, giving a risk factor of 1.3, which is rather high due to the difficulty of the task. The value of the text analysis tool for any single requirement document analysis job is therefore:

€12,000 – (4.000 x 1.3) = €6,800

The total value of the tool is this amount times the number of requirement documents analyzed. This value could of course be improved by reducing the costs of maintenance increasing the tool quality, thus reducing the risk factor.

The comparison of work processes with and without automation is a proven method for deriving the benefits of a software system. Subtracting Costs * Risks from the benefit will give the approximate value of the system. The case study in Section 4.3 demonstrates that value can also be negative, in which case a software system is not an asset but an obligation.

4.2 Estimating the Costs of Maintenance

Estimating maintenance costs has been well covered in the pertinent literature. In the original COCOMO model Boehm suggested the equation:

*AnnualMaintEffort = Systemtype * [*

*(AnnualChangeRate **

*DevelopmentEffort) **

QualityAdjustment]

Sneed later refined this model to apply to individual maintenance projects by coupling the estimation process to an impact analysis of the system under maintenance [24]. As described in Section 3, cost is calculated by making an impact analysis of the proposed change, converting that impact domain into a size metric, and then converting the size metric into effort and cost. The benefit is determined by assessing the value of the change proportional to the value of the software product as a whole and adjusting that value by the relative weight of the altered or added functionality.

The following case study illustrates how the value of a software maintenance project can be assessed using the model presented here.

4.3 The German Social Aid System

There are 8 million recipients of social aid in Germany, which is about 20% of the working population. These recipients get payments of €500 to €800 per month to cover their cost of living plus medical insurance and minimum old age insurance. The software for administering this aide is referred to as ALU-2 [25]. It was originally implemented with Micro Focus COBOL, but in the meantime there are many Java components.

Since it was first released in 2005, the system has caused many problems. The first version was so inflexible that it was not possible to adjust the tariff for medical insurance with the consequence that the state transferred €25 million per month too much to the health insurance companies. In addition every day 1 to 2 working hours were lost per user due to errors and poor usability. It is estimated that users of the social aide agency have to work 15% to 20% more to compensate for the ineffectiveness of the system. There are now 146 known work-around required to use the system.

The German Social Ministry had to admit before a parliamentary investigation that the system was costing yearly €380 million to operate. The cost of administering the social aide without automation was put at less than €300 million. This results in a benefit of minus €80 million per year [24].

The costs of maintaining the ALU-2 system have not been officially announced for fear of public repercussion, but unofficial sources claim there are some 62 persons working to correct errors and to make necessary changes. Considering personnel costs in Germany to be at €200,000 per year, this amounts to an annual maintenance cost of €12.5 million. On top of this comes the risk of 15% that the system will produce a wrong result. Given the rule:

$$Value = Benefits - (Costs * Risks)$$

the value of the social aide software system is:

$$-80 - (12.5 * 1.15) = - €94 \text{ million per year}$$

The provider of the ALU-2 software is now offering to perform a reengineering project to improve the usability, performance, and reliability of the ALU-2 system, as well as to reduce the maintenance costs. It is promised from the provider that the usability will be increased by 25%, the performance by 20%, and the reliability by 33%. Assuming that usability makes up for 20% of the total system benefit, performance for 15%, and reliability for 25%, then the overall increase of benefit would be:

$$(.25 * .2) + (.2 * .15) + (.33 *. 25) = .16$$

This would mean that by reengineering the ALU-2 software the provider could increase the benefit of the system by 16%, thereby decreasing the cost of operating it from €380 million to €319 million. This is still more than the cost of the old manual system, but not so much.

To that comes the reduced cost of maintenance that the provider puts at 20%. Since there are now 62 persons involved in maintaining the system, only 52 should be required in the future. The real big savings will come by transferring the maintenance operation to the provider's new partner in India where the costs of a maintenance engineer are less than €40,000 per annum. With 52 maintainers at €4,000 per year, the costs of maintenance should go down from €12.5 million to €2.5 million, giving a savings of 20% or €10 million per year.

The costs of the reengineering project, which will also be performed by the Indian partner, are estimated at 150 person months. At a cost of €40,000 per man-year, the reengineering project would cost some €6 million. To this comes a risk factor of 1.4 for contingencies. Assuming that the ALU-2 software will live another 5 years before being totally replaced the ROI of the reengineering project is:

$$Benefit = [(80 + 10) * 5] = 450 \text{ million}$$

$$Costs = 6 * 1.4 = 8.4 \text{ million}$$

$$ROI = [450 - 8.4] / 8.4 = 52.5$$

The total savings over the 5-year period should amount to €450 million. The onetime costs of €8.4 million are minimal compared to the savings, thus making this project well worth it. The important factor is less the reengineering than the turning over of the software to the Indian partner. Even if the former German maintainers themselves become social aid recipients, receiving €800 a month it is still cheaper for the German state to move the maintenance of the social aide system to India. As shown in Fig. 1, this example illustrates the fact that labor-intensive projects involving large systems can be significantly cheaper when done offshore – provided the project can be adequately specified and structured. In the case of maintenance and reengineering projects, this should be possible.

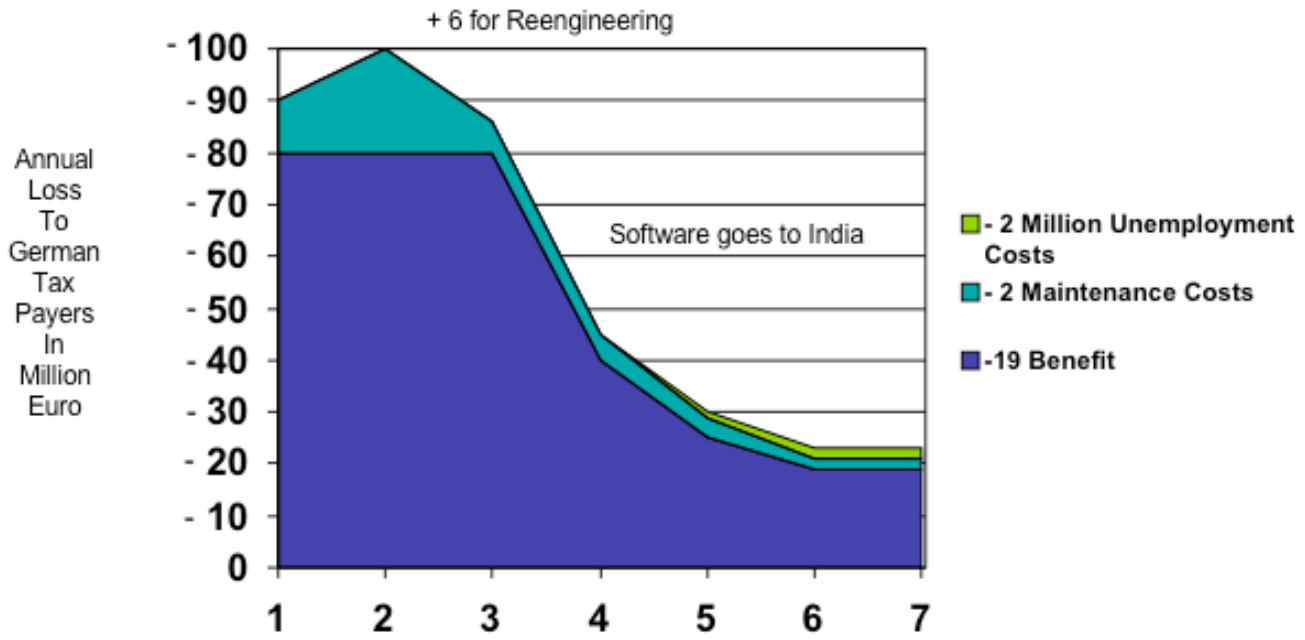

Figure 1: increasing value of ALU-2 software

5 Summary

This paper has attempted to make a case for value-driven software maintenance as an extension of value-driven software engineering. It proposed and demonstrated a methodology for assessing the value of software maintenance and predicting the costs of the maintenance tasks based on impact analysis. To calculate the value of a maintenance project, it is first necessary to determine the value of the software system being maintained. This can be done using Sneed's value equation. Then it is possible to determine the added value produced by each maintenance task as a percentage of the total system value. The value of a new release is the sum of the values for each individual error correction, change, and enhancement.

The value calculation presumes that both costs and benefits can be computed. The benefit of a maintenance task is the difference between having the result of that task and not having it. The costs of a maintenance task can be estimated based on the size of the impact domain adjusted by the complexity and quality of that domain. By comparing the costs of maintenance with the benefits therein, it is possible to calculate the return on investment of maintenance tasks, thus giving maintenance managers a criterion for screening and prioritizing maintenance. A Germany Social Aid system was used as the case study to demonstrate the validity of the proposed methodology.

Conducting software maintenance & evolution is very much needed in leveraging the invaluable assets in legacy systems. Extending value-driven software engineering concepts to software maintenance is beneficial. Much work remains to be done in the future. A next task is to conduct a study comparing other examples with data from historical projects that have been completed to the case study in this paper in order to have a more general conclusion of the efficacy of value-driven software maintenance.

References

[1] B. Boehm, *Software Engineering Economics* (Englewood Cliffs, NJ: Prentice-Hall, 1981, p. 597).

[2] B. Boehm, L. Huang, Value-based software engineering, *IEEE Computer*, March 2003, p. 33.

[3] F. Hayek, *The Maintenance of Capital Profits, Interest and Investment* (London: Routledge & Sons, 1939).

[4] F. Hayek, *The pure theory of capital* (Chicago: University of Chicago Press, 1941, p. 93).

[5] B. Boehm, R. Turner, Management challenges to implementing agile processes, *IEEE Software*, Sept. 2005, p. 30.

[6] T. Little, Value creation and capture – A model of the software development process, *IEEE Software*, May 2004, p. 48.

[7] B. Boehm etc., *Software Cost Estimation with COCOMO-II* (Upper Saddle River, N.J: Prentice-Hall, 2000, p. 57)

[8] S. Biffl , *Value-based Software Engineering* (Berlin : Springer, 2006, p. 21).

[9] R. Solingen, Measuring the ROI of software process improvement, *IEEE Software*, May 2004, p. 32.

[10] Lehman MM and Belady L, Program Evolution – Processes of Software Change, Academic Press, London, 1985.

[11] IEEE: ANSI/IEEE Standard 1219-1993, *Standard for Software Maintenance* (New York: IEEE Press, 1993, p. 15).

[12] N. Chapin, J. Hale, K. Kahn, J. Ramil, W. Tan, Types of software evolution and maintenance, *Journal of Software Maintenance and Evolution*, *13*(1), Jan. 2001, p.3.

[13] H. Sneed, A cost model for software maintenance and evolution, *Proceedings of 20th International Conference on Software Maintenance*, IEEE Press, Chicago, 2004, p. 264.

[14] L. Hatton, How accurately do engineers predict software maintenance tasks? *IEEE Computer*, Feb. 2007, p. 64.

[15] Sneed, H., Impact analysis of maintenance tasks, *Proc. of 17th International Conference on Software Maintenance*, IEEE Press, Florence, 2001, p. 180.

[16] B. Boehm, L. Huang, How much software quality investment is enough – A value-based approach, *IEEE Software*, Sept. 2006, p. 88.

[17] A. Mockus, L. Votta, Identifying reasons for software change using historic databases, *Proceedings of 16th International Conference on Software Maintenance*, IEEE Press, San Jose, 2000, p. 120.

[18] K. Beck, Clean code – pipe dream or state of mind, Smalltalk Report Nr. 4, June, 1995, p. 20.

[19] H. Sneed, Estimating the costs of reengineering projects, *Proceedings of 12th Working Conference on Reverse Engineering*, IEEE Press, Pittsburgh, 2005, p. 111.

[20] S. Tilley, J. Gerdes, T. Hamilton, S. Huang, H. Mueller, D. Smith, K. Wong, On the business value and technical challenge of adopting web services, *Journal of Software Maintenance and Evolution*, *16*(1-2) Jan. 2004, p. 31.

[21] H. Sneed, Economics of Software Reengineering, *Journal of Software Maintenance*, *3*(1), Sept. 1991, p. 163.

[22] H. Sneed, *Softwaresanierung* (Koeln : Rudolf Mueller Verlag,1991, p. 133).

[23] E. von Hahn, *Werterhaltung von Software* (Wiesbaden: German University Press, 2005, p. 19)

[24] H. Sneed, Estimating the costs of software maintenance tasks, *Proceedings of 11th International Conference on Software Maintenance*, IEEE Press, Opio, France, Sept. 1995, p. 168.

[25] Redaktion, ALU-II Pannensoftware vor der Ablösung, *Computer Weekly*, Nr. 11, Feb. 2008.

Authors Biographies

Harry Sneed is a test engineer in Anecon GmbH in Vienna and is a lecturer at the University of Linz in Austria teaching Software Evolution. He has been

working as Programmer/Analysis, Project Leader and Research Consultant in various large-scale industrial projects, including U.S. Navy, Volkswagen Foundation, Hungary, Union Bank of Switzerland, and three ESPRIT projects. He has published more than 180 technical articles both in English and German Science journals and proceedings, and 17 books in German. He received the best Industrial Paper award in ICSM 2002 in Montreal. He was the General Chair of ICSM 2005 in Budapest. He is co-Chair of GI-Fachgruppe for software Management and serves in the GI committee for Metrics, Test and Reengineering. In 1998 he was elected to the technical Committee of the IEEE Computer Society and in 2005 he was appointed to be a fellow of the German Informatics Society.

Shihong Huang is an Assistant Professor in the Department of Computer Science & Engineering at Florida Atlantic University. She has a Ph.D. from the University of California, Riverside. Her research interests include reverse engineering, program comprehension, software systems redocumentation, and software maintenance & evolution. She was the General Chair of the 24th ACM International Conference on Design of Communication (SIGDOC 2006), and is Program Co-Chair of the 9th IEEE International Symposium on Web Site Evolution (WSE 2007).