Software Deterioration And Maintainability – A Model Proposal

Rikard Land
Mälardalen University
Department of Computer Engineering
Box 883
SE-721 23 Västerås, Sweden
+46 (0)21 10 70 35
rikard.land@mdh.se
http://www.idt.mdh.se/~rld

ABSTRACT

In this paper, we connect the notion of software maintainability with the problem of software deterioration. We propose a model that incorporates both of these aspects, which gives us a vocabulary and a more formal tool than before, and allows us to discuss how to maintain software so as not to make it deteriorate.

This paper describes the problem areas of software maintenance and software deterioration, describes the proposed model, and suggests ways of verifying it.

Keywords
Software aging, software architecture, software deterioration, software maintainability, software measurement.

1. TWO PROBLEM AREAS

Have you ever heard of a piece of software that at its initial release was considered perfect and was never subject to modifications? I daresay that the total experience from the software community strongly supports the opposite statement: if a piece of software is at all useful, it will become modified. The activity of modifying software that has once been put into use is called *maintenance*.

It is also a general observation that, as a piece of software ages, it "deteriorates", partly because it is being maintained. In this paper, we describe a model that connects the notions of software maintenance and that of software deterioration by saying that it is important to understand how to maintain software *to keep it maintainable*, i.e. so as not to make it deteriorate.

The importance of constructing maintainable software is illustrated by the common observation that maintenance costs usually grows to more than the initial development cost [30], partly because of its deterioration as changes are made to it [38]. In this paper, we propose a model that captures and relates both of these problems: software maintenance and software deterioration. The model is by no means verified; we describe how it can be verified or falsified through experiments and case studies.

There are many definitions of *software maintainability*, in essence very similar [3,4,16,42]. We here quote the IEEE Standard Glossary of Software Engineering Terminology [15]:

> **maintainability.** […] The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.

The problem of "software deterioration" [3,5,29,37], also called "design erosion" [40], means that as a piece of software ages, and is being maintained, it becomes harder and harder to maintain. Its conceptual integrity [7] has been violated; it has become patched and mended beyond repair.

We have felt that these issues needs to be formalized somewhat by establishing relationships between the terms used, to be able to explain long-term trends. In section 2 we propose a model combining these two notions (software deterioration and maintainability), and section 3 outlines how future research, through experiments and case studies, could support (or falsify) the model.

2. THE MODEL

Before we present the model, we should present the assumptions we build it on.

2.1 Assumptions

We build our model on two basic assumptions, one that is usually implicit in maintainability research literature, and one from literature on software deterioration. From "between the lines" in maintainability texts we can extract the following statement, very much in line with the earlier quote IEEE definition:

> **Assumption 1:** In a system that is easy to maintain, a change can be implemented with less effort than in a system that is harder to maintain (but in other respects equivalent).

And from texts on software aging and deterioration we form our second assumption:

Assumption 2: The typical observation is that software deteriorates because changes are not implemented careful enough; if the changes were implemented more carefully, the system would deteriorate at a much slower rate.

With "not careful enough", we mean that maintainers change the software somewhat careless, either because of time constraints, or due to too little knowledge about the initial concepts of the design, or merely because of, well, carelessness. If we accept these assumptions, we can now turn to the description of the model itself.

2.2 The Model Itself

In the model, we use three dimensions and define the relationships between them: "maintainability", "effort", and "change". We will then use relatively simple mathematics to formulate the relationships between them. Before we continue, let us make some notes concerning these terms:

- **Maintainability.** It is not obvious how to measure "maintainability", but there are suggestions of such measures. We will return to this issue in section 3.1, but for the moment we assume that there is a way to measure "maintainability".
- **Effort.** The FEAST/1 project (Feedback, Evolution And Software Technology) used the number of modules "handled" during a change to approximate "effort" [25]. However, in the FEAST/2 project, this measure was exchanged for the number of programmers involved during each month [32].
- **Change.** The "change" dimension could be thought of as "discrete time", with which we mean that a number of changes are implemented sequentially to a system. If the changes are implemented with a constant interval (say, one per day), the "change" dimension could thus be exchanged for "time".

Let us for a moment assume that there is a way to measure "maintainability" (we will return to this issue in section 3.1). In our model, this means (quite intuitive) that the higher the maintainability, the less effort is required to implement a change. However, we distinguish between two approaches to implement changes: it is possible to choose between making a "fast hack" (the change is implemented with as little effort as possible) or a "controlled update" (to avoid system deterioration). The controlled update by definition requires more than, or at least equal to, the minimum effort to implement the change through a "fast hack" (see Figure 1). If each change is implemented using the "fast hack approach", the system's maintainability decreases over time, while controlled updates ideally preserve it (or even increase it), as Figure 2 describes; indeed, we could even use this observation as a definition and say that a controlled update is "an update using the minimum effort required to preserve the maintainability of the system" (but this would make it easy to forget that it is the long-term effects we want to capture).

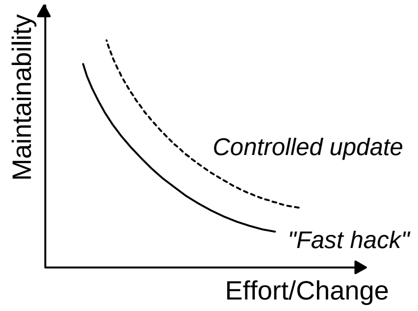

Figure 1. The correlation between "maintainability" and effort required to implement a change[1].

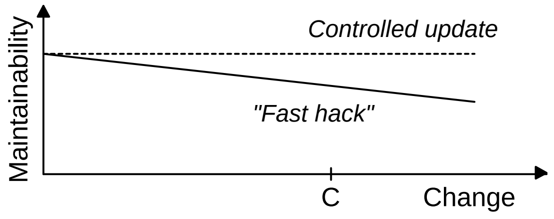

Figure 2. The system deteriorates (measured in maintainability) as changes are implemented.

As the system deteriorates, i.e. its maintainability decreases; we can read from Figure 1 that each new change costs more and more to implement, leading to the graph in Figure 3. If we integrate over "change", we get the graph presented in Figure 4, and it is easy to see that there is a change C at which the accumulated effort for "fast hacks" is equal to the effort for controlled updates; this point is marked in both Figures 2, 3, and 4. We read in the figures that if we have used the "fast hack" approach we have now locked ourselves in a trap: although we have spent the same amount of effort up to change C, the system has a lower degree of maintainability; consequently, each change after C will be more costly than if the system had been updated in a more structured way from the beginning.

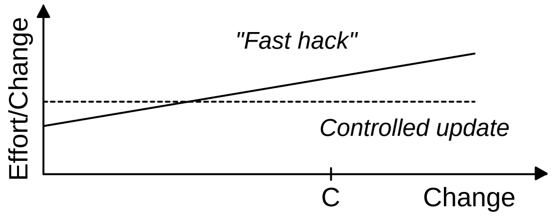

Figure 3. The effort required per change increase as more and more changes are implemented.

[1] It should be noted that the graphs in this section only describe trends and relations ("more than" or "less than"); they are not intended to predict any absolute values, nor are the shapes of the curves by any means exact.

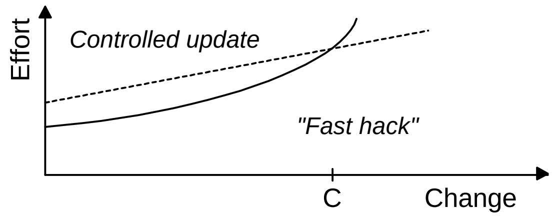

Figure 4. The accumulated effort differs drastically between the two approaches.

We should note that the graphs, as drawn here, are derived from the assumptions; if it is indeed true that the graph of Figure 1 is correct, *and* controlled updates counteract the deterioration of the software (while “fast hacks” rather speed up that process), the other graphs are just mathematical derivations from these “facts”.

However, we must be careful when drawing conclusions: it is e.g. not necessarily true that by spending more effort on a change it is more controlled. Therefore, the model must be used with great care, and measurements supposed to support (or contradict) the model must be analysed thoroughly. We should be careful not to just accept the model: it has to be verified.

2.3 Usability Of the Model

But before turning into how to verify the model, we should ask ourselves of what use the model could be (if it turns out to be a reasonable description of reality); is it worthwhile trying to verify it or would that be just an intellectual exercise? We believe the model is useful since it makes the relationships between the dimensions explicit; we have a tool that can be used for reasoning about e.g. costs. The model provides a foundation for more research on details, like the formulas for maintainability that have been proposed (see section 3.1). Given some quantitative details, in particular about the correlations of Figure 1, the model may be used to answer questions like:

- How do short-term and long-term costs affect each other?
- What is an appropriate level of maintainability?

It may be that a very high level of maintainability (which sounds intuitively appealing) is too costly; this partly depends on the difference between the correlations of Figure 1 – how much more costly is it to do a “controlled update”? However, if it is known that the system will be phased out before the critical change C, it will be advantageous to use the “fast hack” approach. For systems that are expected to live longer than change C, perhaps it is more advantageous to make a series of fast hacks, followed by a restructuring of the system at some well-chosen point in time (before this change C)? Given a good estimation of how costly a restructuring would be, our model could be used as a help when determining when the restructuring should optimally take place, taking the effort required to restructure the system into account. This strategy is shown in Figures 5, 6, and 7.

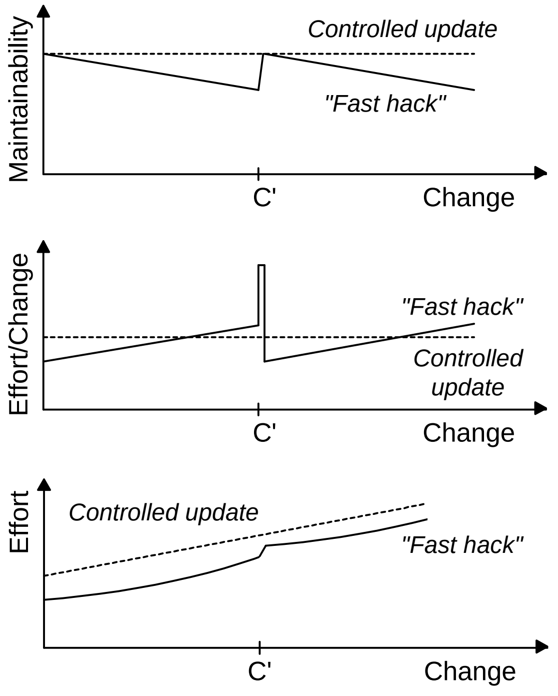

Figure 5, 6, and 7. The accumulated effort can be lower using a series of “fast hacks”.

By introducing this model, we thus hope that it will be easier to decide how to maintain software in a cost-effective way, including dealing the risk of its long-term deterioration. However, we do not claim that we present a solution to this ultimate objective, but rather a tool that supports discussions about how to reach there.

3. HOW TO VERIFY THE MODEL

The model is not verified, but has been constructed through reading between the lines in literature on software maintainability and software deterioration, and by appealing to common sense. In this section we describe how we believe it is possible to verify it using scientific methods, and also somewhat quantify its unknown parameters.

Since we want the results to be of interest for the industry, we would want to verify it in a real industrial environment. The ideal experiment would be using a setting where a series of change requests are implemented, using both “fast hacks” and more controlled updates in parallel (to the same system), while carefully keeping notes on the effort required for each change, and measuring maintainability before and after each change. Hopefully, with a data set large enough, it would be possible to analyse the data and construct the graphs we have described in section 2.2. However, it is not practically possible to perform such an experiment in a real industrial setting; rather, we will have to be content with some type of artificial experiment. It is then important to use a setting that resembles industry as much as possible. After an initial, somewhat artificial experiment, described in section 3.2, we would like to perform case studies, as described in section 3.3.

3.1 Measuring Maintainability

But let us first return to the issue of how to measure maintainability. If we cannot do that, the whole model and any attempt to verify it are in vain.

3.1.1 Code-Level Measures

Fortunately, many researchers have tried to quantify maintainability in different types of measures [1,2,9,27,28,42], of which the most noticeable probably is the Maintainability Index, MI [28,36], which includes McCabe's cyclomatic complexity measure [33]. The Halstead source code measures proposed in the seventies [13,35] have also been used for describing maintainability [35,36]. However, none of these can be said to capture all aspects of maintainability as they focus on the code. As an example, the formula for calculating MI includes the term "average percent of lines of comments per module"; although this is to some extent reasonable it is impossible to know how well the comments describe the code: they might fail to describe the code on a higher level than the code itself, or they may simply be out-of-date, or wrong (the same goes for all documentation, which indeed is important when maintaining software in practice).

We believe that these measures capture important aspects of maintainability, but they are typically validated using expert judgments about the *state* of the software [9,42], while we rather intend to investigate how the measures *change* as the software is maintained. The idea of performing such an evaluation before and after a change is implemented has already been discussed, although the objective of these studies has been to verify cost predictions [2,9,31]; we adopt this approach but pursue it even longer by investigating not only two subsequent changes, but a series of changes, in the case of case studies covering the complete evolution history (very long at least) of industrial systems.

We should make some notes about other proposed "complexity measures": the Function Point measure [10,34], the Object Point measure [10], and DeMarco's specification weight metrics ("bang metrics") [10]. All of these require human intervention (to e.g. grade items as "simple", "average", or "complex") since not all parameters are measurable from source code, which makes them highly impractical when evaluating a large number of subsequent versions; there is also a high risk of mistakes in counting or unfairness in rating. These measures were also designed for cost estimations (before source code is available) rather than of performing measurements on existing code.

3.1.2 Architecture-Level Measures

Our research interests includes software architecture [3,5,14,37] and component-based systems [39]; in this section we therefore discuss the relation between the software maintenance field and that of software architecture. Within the software architecture literature, the terms "maintainability" and "modifiability" are often used informally as a desired feature [3,5,14,37] – indeed, it is one of the very goals with software architecture to make a system understandable, and thus maintainable, by providing abstractions on an appropriate level.

One important goal for maintainability research is to make it possible to accurately estimate maintenance costs; such estimations should be done early in the development, i.e. during the architectural design. Such prediction models are often based both on the argument that maintainability must be discussed in the context of particular changes – one fault may be corrected by changing a few lines of code, while another may require the system to be re-architected. Thus, a system (or component) is always maintainable (or not) *with respect to certain changes*; the use of scenarios to evaluate maintainability has therefore been put forward, particularly on the architectural level [3-6,8,18-21]. The Software Architecture Analysis Method (SAAM) [3,20] and Architecture Trade-off Analysis Method (ATAM) [21] are general scenario-based evaluation techniques with which any quality attributes can be estimated on the architectural level; these have been reported useful in practice [3,19,22]. Bengtsson has suggested one cost estimation model where the type of change (new components, modified components, or new "plug-ins") of each change scenario is taken into account to calculate the estimated change effort [4].

The fields of software architecture and software maintenance clearly have much in common, and in our verification we will therefore investigate how architecture-level measures relates to the code-level measures and our model. There are not as many measures proposed on the architectural level, but the most obvious aspect to investigate is the interdependencies between components. We have in literature found some variants of the number of calls into and number of calls from a component [24], also called "fan-in" and "fan-out" measures [12]. But it has been pointed out that such measures are not as simple as it may first look: from the maintainability point of view there is e.g. a great difference from a function call with only one integer parameter and one with many complex parameters; one must also consider to what extent we are interested in unique calls (to not penalize reuse). These issues are thoroughly discussed in [11]. In the FEAST projects, the researchers investigate the number of "subsystems" handled (i.e. changed, added, or deleted) at each change [25,26,31,32].

3.2 Experiment

Some volunteers are gathered, possibly Computer Science students (who can be enticed into almost anything to an affordable price). They are initially assigned the task of studying and understanding a system, and are then given a set of identical change requests; half of the subjects are told to implement each change as fast as possible, while the other half are told to make the change as controlled as possible. After each change, the software is checked in into a revision control system so that we can perform measurements on the source code (see section 3.1). Of course, the time required to implement each change is carefully recorded. Through this setting, we will gather a set of data which reveals the relationships between the dimensions "Maintainability", "Effort", and "Change"; hopefully such an experiment should be able would be able to give an initial indication, or rule of thumb, of the correlation between "Maintainability" and "Effort/Change" for the two different approaches. The experiment would thus be a quantitative refinement of the now only qualitative model (which is not even verified).

There are of course additional details to be considered when designing and performing the experiment [41]. For example, to avoid the threat that one of the groups turn out to be much better than the other, the experiment should be counter-balanced. This means that we use two different systems and repeat the experiment, but let the groups switch "treatments", i.e. the group that are told to use the "fast hack" approach on the first system are told to do "controlled updates" to the second. That is:

- System 1:
  - Group 1 gets Treatment 1
  - Group 2 gets Treatment 2
- System 2:
  - Group 1 gets Treatment 2
  - Group 2 gets Treatment 1

It may be that the collected data does not look at all as we have sketched in the figures, and we must then find reasons for this. Perhaps our experiment did not reflect the full complexity of the problem: our volunteers might have been too inexperienced in carrying out controlled maintenance tasks (as opposed to fast hacks, which they might be used to from course assignments), they might have had too little understanding of the software where supposed to maintain, the systems might have been misrepresentative, or the set of change requests might have been too artificial; we have to make everything we can to eliminate such error sources. Or, perhaps either the model or the assumptions were simply wrong: if the experiment seems to contradict the model (or the assumptions), this would be an interesting starting point for future research to explain why. But if the results of such an experiment are encouraging, we believe that there is more to learn from industrial case studies, even though these are much less controlled.

3.3 Case Studies

We have been promised access to several large industrial software systems, where all earlier revisions are available together with change descriptions for each revision. This allows us to validate the experimental results (performed in a more or less artificial setting) in a real industrial situation. However, although we will be able to accurately plot the "maintainability" and "change" dimensions, it will be substantially harder to know the effort spent on each change. And we will not be able to know whether a "fast hack" approach was used or the maintainer believed enough time was spent to make the update controlled, since there is no parallel system to compare with. However, we might find rules of thumb from the experiment that can be tested in such an industrial case study. We might for example find changes performed that seem to be "fast hacks" or "controlled updates", and through interviews or questionnaires find whether the maintainers agree or not, to validate the model's predictions. We also intend to investigate whether it is possible to use the data gathered in the FEAST projects [25,26,31,32].

Just by measuring how the maintainability of the system has changed throughout its history, we can easily describe whether the system has deteriorated or not (measured in maintainability, as our starting point have been). Maybe we will find changes that contradict the general trend, which is an interesting for more thorough analysis. A similar investigation has been carried out before, but only a few versions of a rather small system were used [17]. We have presented this objective earlier [23], but not yet carried out the measurements. We believe that such case studies will benefit from performing an experiment beforehand, and therefore intend to carry out the experiment first.

3.4 Valid Verification?

We clearly face a delicate problem: our model could be used as one type of verification for maintainability measures, but we will have to verify it using these measures – there seems to be a sort of circular verification. However, these measures have been shown to make at least some sense, so with data sets large enough, it might be possible to achieve a sort of mutual support for the model and the measures, which need not be scientifically wrong: if both the measures and the model intuitively make sense and they can be shown to support each other, we can be more confident in both.

And we should not forget that any attempt to describe complex realities in simple models or formulas always must be approximate and never universally valid. This goes for the existing maintainability measures we have described as well as for the model we propose.

4. CONCLUSION AND FUTURE WORK

We have proposed a model that connects the notion of software maintainability with the problem of software deterioration. The model gives us a vocabulary and a tool that allows us to discuss how to maintain software so as not to make it deteriorate. We also noted that although we believe it appeals to common sense, the model is by no means verified. We have described how experiments and case studies could support (or falsify) the model, and yield approximate values of the unknown parameters of the model.

5. ACKNOWLEDGEMENTS

I would like to thank professor Ivica Crnkovic and my fellow Ph.D. student Andreas Sjögren for the discussions that led to this paper.

REFERENCES

[1] Aggarwal K. K., Singh Y., and Chhabra J. K., "An Integrated Measure of Software Maintainability", In *Proceedings of Annual Reliability and Maintainability Symposium*, IEEE, 2002.

[2] Ash D., Alderete J., Yao L., Oman P. W., and Lowther B., "Using software maintainability models to track code health", In *Proceedings of International Conference on Software Maintenance*, IEEE, 1994.

[3] Bass L., Clements P., and Kazman R., *Software Architecture in Practice*, Addison-Wesley, 1998.

[4] Bengtsson P., "Architecture-Level Modifiability Analysis", Ph.D. Thesis, Blekinge Institute of Technology, Sweden, 2002

[5] Bosch J., *Design & Use of Software Architectures*, Addison-Wesley, 2000.

[6] Bosch, J. and Bengtsson, P., An Experiment on Creating Scenario Profiles for Software Change, report ISSN 1103-1581, Department of Software Engineering and Computer Science, University of Karlskrona/Ronneby, 1999.

[7] Brooks F. P., *The Mythical Man-Month - Essays On Software Engineering, 20th Anniversary Edition*, Addison-Wesley Longman, 1995.

[8] Clements P., Kazman R., and Klein M., *Evaluating Software Architectures: Methods and Case Studies*, Addison Wesley, 2000.

[9] Coleman D., Ash D., Lowther B., and Oman P., "Using Metrics to Evaluate Software System Maintainability", *IEEE Computer*, volume 27, issue 8, 1994.

[10] Fenton N. E. and Pfleeger S. L., *Software Metrics - A Rigorous & Practical Approach*, PWS Publishing Company, 1997.

[11] Ferneley E.H., "Design Metrics as an Aid to Software Maintenance: An Empirical Study", *Journal of Software Maintenance: Research and Practice*, volume 11, issue 1, 1999.

[12] Grady R.B., "Successfully Applying Software Metrics", *IEEE Computer*, volume 27, issue 9, 1994.

[13] Halstead M. H., *Elements of Software Science, Operating, and Programming Systems Series Volume 7*, Elsevier, 1977.

[14] Hofmeister C., Nord R., and Soni D., *Applied Software Architecture*, Addison-Wesley, 2000.

[15] IEEE, IEEE Standard Glossary of Software Engineering Terminology, report IEEE Std 610.12-1990, IEEE, 1990.

[16] ISO/IEC, Information technology - Software product quality - Part 1: Quality model, report ISO/IEC FDIS 9126-1:2000 (E), ISO, 2000.

[17] Jaktman C. B., Leaney J., and Liu M., "Structural Analysis of the Software Architecture - A Maintenance Assessment Case Study", In *Proceedings of The First Working IFIP Conference on Software Architecture (WICSA1)*, Kluwer Academic Publishers, 1999.

[18] Kazman R., Abowd G., Bass L., and Clements P., "Scenario-Based Analysis of Software Architecture", *IEEE Software*, volume 13, issue 6, 1996.

[19] Kazman R., Barbacci M., Klein M., and Carriere J., "Experience with Performing Architecture Tradeoff Analysis Method", In *Proceedings of The International Conference on Software Engineering, New York*, 1999.

[20] Kazman R., Bass L., Abowd G., and Webb M., "SAAM: A Method for Analyzing the Properties of Software Architectures", In *Proceedings of The 16th International Conference on Software Engineering*, 1994.

[21] Kazman R., Klein M., Barbacci M., Longstaff T., Lipson H., and Carriere J., “The Architecture Tradeoff Analysis Method”, In *Proceedings of The Fourth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, IEEE, 1998.

[22] Land R., “Improving Quality Attributes of a Complex System Through Architectural Analysis - A Case Study”, In *Proceedings of 9th IEEE Conference on Engineering of Computer-Based Systems (ECBS)*, IEEE, 2002.

[23] Land R., “Measurements of Software Maintainability”, In *Proceedings of ARTES Graduate Student Conference (neither reviewed nor officially published)*, ARTES, 2002.

[24] Lanning D.L. and Khoshgoftaar T. M., “Modeling the Relationship Between Source Code Complexity and Maintenance Difficulty”, *IEEE Computer*, volume 27, issue 9, 1994.

[25] Lehman M. M., Perry D. E., and Ramil J. F., “Implications of evolution metrics on software maintenance”, In *Proceedings of International Conference on Software Maintenance*, IEEE, 1998.

[26] Lehman, Meir M. and Ramil, Juan F., *FEAST project*, URL: http://www.doc.ic.ac.uk/~mml/feast/, 2001

[27] Oman P. and Hagemeister J., “Metrics for Assessing a Software System's Maintainability”, In *Proceedings of Conference on Software Maintenance*, IEEE, 1992.

[28] Oman, P., Hagemeister, J., and Ash, D., “A Definition and Taxonomy for Software Maintainability”, report SETL Report 91-08-TR, University of Idaho, 1991.

[29] Parnas D. L., “Software Aging”, In *Proceedings of The 16th International Conference on Software Engineering*, IEEE Press, 1994.

[30] Pfleeger S. L., *Software Engineering, Theory and Practice*, Prentice-Hall, Inc., 1998.

[31] Ramil J. F. and Lehman M. M., “Metrics of Software Evolution as Effort Predictors - A Case Study”, In *Proceedings of International Conference on Software Maintenance*, IEEE, 2000.

[32] Ramil J. F. and Lehman M. M., “Defining and applying metrics in the context of continuing software evolution”, In *Proceedings of Seventh International Software Metrics Symposium (METRICS)*, IEEE, 2001.

[33] SEI Software Technology Review, *Cyclomatic Complexity*, URL: http://www.sei.cmu.edu/, 2002

[34] SEI Software Technology Review, *Function Point Analysis*, URL: http://www.sei.cmu.edu/, 2002

[35] SEI Software Technology Review, *Halstead Complexity Measures*, URL: http://www.sei.cmu.edu/, 2002

[36] SEI Software Technology Review, *Maintainability Index Technique for Measuring Program Maintainability*, URL: http://www.sei.cmu.edu/, 2002

[37] Shaw M. and Garlan D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.

[38] Sommerville I., *Software Engineering*, Addison-Wesley, 2001.

[39] Szyperski C., *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[40] van Gurp J. and Bosch J., “Design Erosion: Problems & Causes”, *Journal of Systems & Software*, volume 61, issue 2, 2002.

[41] Wohlin C., Runeson P., Höst M., Ohlsson M. C., Regnell B., and Wesslén A., *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers, 1999.

[42] Zhuo F., Lowther B., Oman P., and Hagemeister J., “Constructing and testing software maintainability assessment models”, In *Proceedings of First International Software Metrics Symposium*, IEEE, 1993.