UNIVERSITY OF JYVÄSKYLÄ

SOFTWARE MAINTENANCE COST ESTIMATION AND MODERNIZATION SUPPORT

ELTIS-project

Version: 1.61	Classification:
Authors: Jussi Koskinen,	Date: 19.6.2003
Henna Lahtonen, Tero Tilus	Status: Final version

CONTENTS

1 INTRODUCTION

This report describes the theoretical background studies of ELTIS (*Extending the Lifetime of Information Systems*) project, based on the preliminary objectives set to the project. ELTIS is concerned with software maintenance, legacy systems, software lifetime, software renewal/modernization support, and relevant decision criterias for software modernizations.

Chapter 2 provides a general introduction to software maintenance and motivates the economic importance of the area. Chapter 3 represents the usually applied classification of software maintenance activities. Chapter 4 charts the area of general software cost models, which potentially may provide a basis for maintenance cost estimations. Chapter 5 is the largest part of the report and deals with software maintenance cost/effort determination. References to main theoretical works are provided. Chapter 6 provides a framework for software modernization and its support techniques. Chapter 7 summarizes the conclusions.

This report refers to the contents of the individual studies deemed as most relevant to ELTIS. The applicability, constraints, validation, maintenance cost drivers, and suggested further research areas of those studies are analyzed and explicated. In addition, there exists many other articles whose reference information is provided in separate ELTIS-bibliography.

2 SOFTWARE MAINTENANCE

Usually software maintenance is defined as changes to software after its delivery to customers. The main process of maintenance is changing of source code. Changing of source code, naturally, also is important in other latter phases of actual system development. The problems with successfully handling large source code masses, however, typically are more severe in maintenance phase. Many of the large maintenance tasks also require versatile skills and thus although maintenance often does not require much innovations, it is actually quite demanding. In an ideal situation the existing code could be reused (Basili, 1990; Rombach, 1991) and modified as flexibly as possible. That would make it possible to retain at least part of the original investment of system development during long lifetime of the system.

The evolution of software is an empirically relatively weakly studied area. One of the main references is Lehman & Belady (1985), which announces the so-called laws of Lehman. A characterization of maintenance which emphasizes configuration management and program comprehension is represented by von Mayrhauser (1994), who has distinguished herself especially in the area of program comprehension. Most of the text-books in the software maintenance area are old, classic ones being Martin (1983) and Swanson & Beath (1989). More recent ones include Takang & Grubb (1996) and Polo *et al.* (2003). Pigoski's (1996) book is practical but remains at rather general level.

2.1 Main empirical studies

2.1.1 Lehman et al. (1998)

This paper describes a subset of the results obtained to the date of publication (1998) from FEAST/1 project and implications on Lehman's laws (listed in appendix 1 of the paper). Two simple metrics of system evolution (*size of system*, *fraction of system not touched at each release*) were observed as functions of *release serial number* and implications on general system evolution and maintenance were drawn from them.

Results suggested a minor rewording of Lehman's 5th law and increased confidence in the validity of the laws. Results also provided significant support for the FEAST hypothesis (formulated in the preprints of the three FEAST Workshops, see http://www.doc.ic.ac.uk/~mml/feast1/).

E-type systems (which are central in the definitions of Lehman's laws) mean applications for "real-world" purposes, which are "connected" to a multi-layered and iterative "feedback system". Positive feedback tends to increase software size, whereas negative feedback tends to stabilize program development.

Lehman's laws are as follows (as formulated in this paper):

1) *Continuing Change*: E-type systems must be continually adapted or they become progressively less satisfactory.
2) *Increasing Complexity*: As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.
3) *Self Regulation*: Global E-type system evolution processes are self regulating.
4) *Conservation of Organisational Stability*: The *average effective global activity rate* in an evolving E-type system tends to remain constant over product lifetime.
5) *Conservation of Familiarity*: On average, the *incremental growth* tends to remain constant or to decline.
6) *Continuing Growth*: The functional content of E-type systems must be continually increased to maintain *user satisfaction* over their lifetime.
7) *Declining Quality*: The quality of E-type systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
8) *Feedback System*: E-type evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such to achieve significant improvement for other than the most primitive processes.

Validation of the results

Data from the evolution of OS/360 (25 releases), ICL VME Kernel (30 releases) and Lucent Technology System 1 (17 releases) and System 2 (14 releases) were used.

3 SOFTWARE MAINTENANCE TASKS

Generally software maintenance tasks are classified into corrective, adaptive, perfective, and preventive categories. The first three of these categories were originally derived from the landmark empirical study of software maintenance by Lientz & Swanson (1980) (covering 487 organizations).

The contents and characteristics of the classes are as follows:

- *Corrections* relate to the diagnosis, localization, and actual fixing of errors. Debugging and testing relate intimately to this class. Often correction-type tasks are easiest and thus less cost-producing ones, but such that they have to be performed within rigid time-bounds.
- *Adaptive tasks* deal with interfacing existing software to changing (technical) environment.
- *Perfective tasks* is the largest category. Additions, enhancements and modifications are made to the code based on (generally often) changing user needs.
- *Preventive maintenance* aims at enhancement of future maintainability of the system. Preventive maintenance is least acute, but because of constant cumulation of maintenance costs, preventive maintenance should be considered in case of software which has long lifetime. Thus it may be a cost-effective strategy in long run.

A much more fine-grained classification of maintenance tasks has been represented more recently by Chapin *et al.* (2001). Maintenance also includes such central (generic) tasks as configuration management, change control, code changes, code localizations, program comprehension and impact analysis. The line of empirical studies of software maintenance processes in case of industry-level software is thin (look *e.g.* von Mayrhauser & Vans, 1995; Singer, 1998; Seaman, 2002).

4 GENERAL SOFTWARE COST ESTIMATION MODELS

There exists established ways to determine the effort needed in software development projects. The results received from the application of the these general models, however, generally are not accurate, although best of them provide relatively good estimates.

In traditional software cost models, costs are derived simply based on *required effort* (which is measured in man-months). Empirical estimation models provide formula for determining the effort based on statistical information (a project database) about more or less similar projects. The precise software development situation is taken into account by using the so-called *complexity factors*. Empirically derived co-efficients are provided in the tables of the models, which take into account the effect of possible deviations from the nominal case. Models usually require *calibration* to the actual software development process of the organization.

COCOMO (Boehm, 1981) is the best known of these models. Boehm states that COCOMO's intermediate model provides estimates which deviate from the actual needed effort (only) about 20% in average. COCOMO-II (Boehm *et al.*, 2000) is a new updated version of the classic model, with a more modern project database. Boehm (1981) has also represented a simple linear model for determining maintenance costs, but he admits that it has many limitations. Also Phua (2002) has given general formulas for maintenance costs.

Other software cost models include those represented *e.g.* by Kitchenham & Taylor (1984). Techniques may be classified into the following classes:

- Analogy-based methods (*e.g.* SSM, LATURI).
- Algorithmic methods (*e.g.* Halstead).
- Composition-based methods, most notably FPA (Function Point Analysis, Albrecht & Gaffney, 1983) (and its variants: Jones, Symons and Reifer).
- Complexity-based (*e.g.* cyclomatic complexity, McCabe, 1976; Gill & Kemerer, 1991) methods.
- Statistics-based methods (*e.g.* Prize-S).
- PERT.
- Putnam's model.
- SOFTCOST.
- Other models (*e.g.* Farr-Zagorsky, NADC, Daly) which are tailored to the needs of specific application areas, such as financing or aviation.

4.1 Main studies

4.1.1 Boehm (1984)

This article summarizes the 1984 state of decision making and software cost estimation techniques. Boehm compares *Algorithmic*, *Expert*, *Analogy*, *Parkinson*, *Price-to-Win*, *Top-Down* and *Bottom-Up* software cost estimation techniques (which have been developed

earlier by various authors) and surveys *decision making under uncertainty.* These techniques are covered in detail in Boehm's classic book (Boehm, 1981).

Parkinson and *Price-to-Win* are judged unacceptable due to the high probability of ending up to results that lead to hazardous decisions. None of the rest is better than the others in all aspects. Boehm suggests that in practice a combination of applicable techniques should always be used, results compared and iterated when they differ.

Boehm presents a "master key to software engineering economics decision analysis techniques": *decision diagram*. Main economic analysis techniques available to support decision making under uncertainty are the following.

In complete uncertainty situation the following may be applied: *maximax*, *maximin* and *Laplace rules*. These, however, are inadequate for practical software engineering decisions.

Expected-value techniques, where probabilities of different outcomes are estimated, and *complete expected payoff* is calculated as follows:

$$E = \sum_{o \in \text{Outcomes}} \Pr(o)\,\text{Po}(o)$$

where E = expected payoff,
$\Pr(o)$ = probability of o,
$\text{Po}(o)$ = payoff if o occurs,
and decision is made using expected payoffs.

Uncertainty may be reduced by "*buying information*". Boehm suggests 5 conditions, titled "*value-of-information guidelines*", under which it makes sense to buy information (by, for example, prototyping). There exists attractive alternatives whose payoff varies greatly, depending on some situation-dependent uncertainty factors. These uncertainty factors have an appreciable probability of producing unwanted outcome (low or negative payoff). The investigations have a high probability of accurately identifying the occurrence of the possible unwanted outcomes. The required resources of the investigations do not exceed the *value of the information* they produce. There exists significant side benefits (*e.g.* team-building, customer relations) derived from performing the investigations.

Applicability of the approach
Given "value-of-information guidelines" and "master key to software engineering economics decision analysis techniques" are directly applicable in a decision making situation. They both translate the known limitations and constraints of decision making strategies to algorithmic form.

Constraints of the approach
There are no explicated constraints.

Validation of the results

This paper is a summary of techniques without detailed descriptions or validation.

Suggested future work
Software data collection is suggested to be performed. A fundamental limitation to progress in software cost estimation is the lack of unambiguous standard definitions for software data (such as: *man-months*, *delivered instructions*).

"The software field cannot hope to have its Kepler or its Newton until it has had its army of Tycho Brahes" Barry W. Boehm.

Other related works
These include Boehm & Papaccio (1988); Sommerville (1996); Bennett & Gittens (1997).

4.1.2 Kemerer (1987)

Kemerer is an acknowledged authority on the field of software metrics. He has discussed the importance of validating general empirical software cost models. Because of the preliminary state of maintenance cost estimation research, the need for empirical data on software maintenance is even clearer.

In this paper four popular algorithmic cost estimation models: SLIM, COCOMO, Function Points Analysis (FPA) and ESTIMACS were evaluated. Generalizability, performance of LOC and non-LOC based models and relation between proprietary and open models were the main targets of research effort.

LOC-based models perform poorly when used in different environment than in which they were developed. Average error rates between the received cost estimates and actual outcomes ranged from 85% to 772% with most estimates having error rate higher than 500%. Thorough calibration with a data from previous projects is a necessity.

Non-LOC models (FPA, ESTIMACS) did generally a slightly better job. In terms of regression analysis results LOC models (COCOMO, SLIM) had higher correlations. However, LOC data used was obtained *ex post*, which is – of course – accurate. In reality cost estimation is based on *ex ante* LOC counts which are nothing more than a "civilized guess".

No conclusive answer to question "Do proprietary models perform better?" could be given, because SLIM outperformed COCOMO, and FPA did somewhat better than ESTIMACS.

Applicability of the approach
This evaluation clearly points out the fact which model developers themselves have heavily underlined: These models are *adjuncts* to, not substitutes for a detailed estimate done by project managers. At the very best, algorithmic models explain 88% of the behavior of the actual man-month effort.

Constraints of the approach
No special constraints in applying the results from this research were specified.

Validation of the results
Validation was done against data collected from 15 large business data-processing projects written mainly in COBOL. Average size of project was 221 KLOC.

Suggested future work
More information is suggested to be acquired on what impact do the estimates themselves have on project.

4.1.3 Grady (1994)

The author classifies uses of software metrics related to the following aspects:
- Project estimation and progress monitoring.
- Evaluation of work products.
- Cyclomatic complexity.
- Design complexity.
- Process improvement through failure analysis.
- Project defect patterns.
- Software process defect pattern.
- Experimental validation of best practices.

He also discusses the usefulness of these categories for engineers, project managers, process groups and higher management. In conclusion part of the article, the author lists the following recommendations for strategic purposes:
- Measures of success should be defined early.
- Data defect trend is useful in release decision.
- Complexity should be measured targeting design decision optimization and to creation of more maintainable product.
- Defects should be categorized (this helps in identifying product and process weaknesses).
- Data which would quantify the success of best practices should be collected.

Following atributes (data which should be collected in a software project) were used in the research: *engineering effort by activity*, *data size*, *defects*, relevant *product metrics*, *complexity* and *testing coverage*.

Applicability of the approach
General results need to be refined for each project. In other words, attributes, their timely analysis and effect on success should be considered individually in each project/organization. The article presents major uses of software metrics and proposes (listed above) attributes, which should be collected in project, which is useful.

Constraints of the approach
No specific constraints, process metrics are suggested to be collected.

Validation of the results
Few examples without exact details are presented from Hewlett-Packard's projects.

Suggested further research
The author claims that more effort should be reserved in research of new practices and their benefits.

Other related works
Fenton (1994); Henry *et al.* (1996); Schneidewind, N. (1997); Bitman (1999); Ramil & Lehman (2000); Pressman (2001).

4.1.4 Briand et al. (2000)

Main goals of this research were to evaluate existing cost models with large data set and to compare *local cost models* to models based on *multi-organizational data.* Selected modeling techniques to be evaluated by MRE (Magnitude of Relative Error) model, PRED(l) (Prediction at level l) test and cross validation were: Ordinary Least-Squares regression (OLS regression), Stepwise ANOVA, Analogy, CART, and combinations of these techniques.

Results indicate that OLS regression and ANOVA performed better than other evaluated techniques. However, consistently with previous research, also the best models proved to be inaccurate. Remarkable differences between local cost models and general cost models weren't found.

Variables which were used in the research (and which potentially are also software maintenance cost drivers) included the following: the *domain* the system was developed for, *adjusted KLOC*, *effort* of project, *team size* at any time, *virtual machine volatility, required reliability, execution time constraints, main storage constraints, programming practices, software tools* and *programming language experience*.

Applicability of the approach
The research is an accurate and repeatable evaluation of cost modeling techniques. Results give us hints, what cost models should we examined and developed.

Constraints of the approach
No explicated constraints.

Validation of the results
The researchers used data from European Space Agency (ESA) multi-organization software project database, which, at the time of the research, contained 166 projects from 69 organizations.

Suggested future work

More research effort is suggested to be directed into studying *subjective effort estimation*, modeling based on *expert knowledge elicitation* and combining techniques of expert opinion and project data.

Other related works

Briand *et al.* (1999).

5 SOFTWARE MAINTENANCE COST ESTIMATION

Software maintenance is clearly the most expensive and laborious phase of system development. Often in case of successful software it causes 50-75% of the costs of system development to the producing organization (Sommerville, 1996, p. 660). The relative importance of maintenance is especially great in case of systems which have long lifetime (*i.e.* legacy systems, look *e.g.* Bisbal *et al.*, 1999) and which are large, complex and critical to the customers.

Generally it is assumed that new design methodologies would alleviate also the problems of maintenance. However, despite the adoption of new design methods, the relative amount of maintenance costs has in fact increased (Edelstein, 1993). Thus maintenance problems appear to be constant problems in organizations producing software, without any imminent, highly effective panacea.

Most of the time used to software maintenance is spent on program comprehension. Since work-time is expensive this underlines the importance of improving the maintenance process, *e.g.* by using up-to-date CASE-tools, such as reverse engineering or reengineering tools. Since cost-effective support tool development requires the identification of the processes which take most of the maintenance time, this is an important related question.

Because of the great ratio of costs caused by maintenance, it would be desirable to estimate the maintenance costs systematically. Their reliable estimation, however, is hampered by the fact that most of the established cost models, such as COCOMO (Boehm, 1981; Boehm *et al.*, 2000) do not fit well to the peculiarities of the maintenance phase. Thus, estimates often are only "enlightened guesses".

A good example of the importance of maintenance problems and problems of their estimation is the Y2K-bug (Feiler & Butler, 1999), which is said to be the single most expensive technical problem in the history. The gap between many of the announced estimates (*e.g.* Jones, 1997) and actual real costs was wide. Generally, the Y2K-bug was successfully remedied. However, the required preventive code correction demanded relatively big effort, and *e.g.* Nokia used about 75 million Euros to that purpose (ITV, 2000).

It would be desirable to be able to make well-informed and correct decisions regarding whether to continue or not to continue maintenance of a peculiar software system (or its part). Continued maintenance potentially enables extension of the system's lifetime, which may or may not be desirable depending on the interests of the organization producing the software and its customers.

In reality there exists a great amount of factors which should be considered. Most of the theoretical models represented earlier make crude simplifications regarding the actual decision

situation. Thus it would be important to collect maximal data-set of the potentially relevant factors.

5.1 Main studies

5.1.1 Sneed (1995a)

This paper proposes ways (and presents an implemented method *Softcalc*) to extend current cost estimation methods to cover the estimation of *maintenance costs.*

Softcalc is carried out in the following 7 steps:

1) *Size*, *complexity* and *quality* of the software are measured (automatically, using code auditor).
2) Impact domain of the planned maintenance action is determined.
3) The *size of the impact domain* is measured, in at least two of the following metrics: *LOC*, *number of program statements*, *function-points*, *data-points*, or *object-points.*
4) The size measure (step 3) is adjusted by a complexity factor (step 1).
5) The size measure is adjusted by the *external* and *internal quality factors* (step 1, internal quality reflects *maintainability*).
6) The size measure is adjusted by a *productivity influence factor* depending on the estimation method used.
7) Adjusted size measure is transposed into *maintenance effort* by means of a productivity table

Applicability of the approach
The model gives one of the inputs (estimated *maintenance costs*) to the decision making process when deciding whether to rewrite current system or not.

Constraints of the approach
Supporting tools are a necessity. Softcalc appears to be well-defined only by the (Sneed's proprietary) tools implementing it. Even if this would not be the case, it would be far too laborious to use it without proper supporting software providing required metrics.

Validation of the results
Sneed gives an example, but no actual empirical validation of any kind. In fact he presents validation as "what is needed".

Suggested future work
According to Sneed, empirically founded correlations between maintenance effort and size, complexity and quality metrics would be interesting. Adequate means of defining the *impact domain* of planned modifications is also interesting. Once this has been solved the scope of

the maintenance action is measurable. Also maintenance cost drivers are interesting: the product (software) itself, product (and business) environment and maintenance personnel.

5.2 Empirical methods

5.2.1 Kemerer & Slaughter (1999)

The article's discussion is focused on *empirical software maintenance research tasks and methods*. In the first section, the authors have summarized results found and methods used in a few prior research articles.

The authors collected data from a large US retailer with centralized information systems department, separate development and maintenance units and great stability of personnel. Research included logs of 23 business systems written in COBOL, but only two systems are compared in the article. Data was categorized into 3 types (corrections, adaptations and enhancements) and processed with time series analysis, sequence analysis, phase mapping, gamma analysis and gamma mapping.

At the end of the article, authors state, that many of the problems of maintenance derives from a lack of knowledge of *maintenance process* and of relationships between *software practices* and *maintenance outcomes*.

They emphasize the importance of participation of a good commercial partner in empirical research projects. Main criteria for a good partner are: 1) a *large data source* from programs and versions and 2) *willingness to cooperate* with the research. Another success factor is a *highly disciplined research approach* with desire to expand the previous research.

Following attributes were used in data collection of this study: *age*, *LOC*, *FP*, *number of modules* (online and batch), *average module size*, *cyclomatic complexity per LOC*, *operators per LOC* (unique and total), *unique operands per LOC* (unique and total), *average cost per change/FP/LOC*, *enhancement cost per enhancement/FP/LOC* and *maintenance cost per maintenance/FP/LOC*.

Applicability of the approach
The article reports guidelines for empirical research and therefore it is useful to us.

Constraints of the approach
Results are possibly restricted to the studied software systems.

Validation of the results
Empirical data of over 25,000 change events was collected from 23 commercial software systems over a 20-year period. However, only two of the systems are reported in the article.

Suggested further research
Authors recommend further research based on their data. Listed examples included:
- Comparing the evolution of legacy systems developed with different approaches.
- Comparing evolution patterns from different kind of industries and organizations.
- Examining the data collected by the authors with other methods to predict the occurrence of *evolution patterns*.

Related works
Kitchenham *et al.* (2002).

5.3 Software lifetime and rewriting strategies

5.3.1 Gode et al. (1990)

A model to compare *rewriting strategies* (*i.e.* should we use the current or some new technology) and determine the optimal rewriting points is represented. The given model is explicit, well-formed and has only three relatively general assumptions:
1) Maintenance cost is convex in amount of maintenance performed (the effect of refactoring is not taken into account),
2) Maintenance cost is decreasing in *system structuredness*,
3) Maintenance cost convexity is decreasing in relation to system structuredness.

Applicability of the approach
All the propositions made, are proven starting from the assumptions listed earlier. Model is very simplistic and probably doesn't contain the most important factors defining rewriting policies. The only apparent, but quite severe, limitation is *finite fixed planning horizon*, which is in fact fourth (hidden) assumption.

The model is a formal exercise, not an applicable model. Although (if possible) examining the results from our research against the propositions of presented model could be interesting.

Constraints of the approach
There exists a huge gap between the simplicity of the model (instantaneous rewriting, and fixed planning horizon are assumed, and there is only a small number of parameters) and versatility of "real life".

Validation of the results
No validation has been performed.

Suggested future work
In conclusion authors suggest: "The next step will involve empirical testing of some of the results obtained in this phase." Also *lifetime drivers*, *technology structuredness* (level of

abstraction in programming languages and environment used to develop and maintain the software), and *maintenance backlog* are suggested to be studied.

5.3.2 Foster (1991)

Widely held views on software maintenance expenditures are examined in this paper. Foster argues that in the light of then recent evidence they must be modified. The "popular view" is summed up as: Software maintenance costs are high and rising as a proportion of software expenditure (commonly referred as the *S-curve*). This is an undesirable situation.

The goal of process improvement in maintenance should be the reduction in the *proportion of maintenance* within software expenditure. However, actual effective maintenance costs collected from various publications (1976-1990) clearly show that the proportion of maintenance costs should be considered to be constant (not rising) over time.

The results of the paper show that if the maintenance process is improved, the results of the effort are to be seen in increased "lives" of maintained programs. Interestingly, there is very little existing data on the software system lifetimes.

Applicability of the approach
Not directly applicable.

Constraints of the approach
Results apply only to data processing systems. Embedded systems are excluded. The economic model referred in the article doesn't take into account the situation, in which increase in development productivity and maintenance productivity occur at the same time.

Validation of the results
This work is based on a literature survey. Maintenance effort data was gathered from the following publications: Bennett *et al.* (1980), Moreton (1988), Foster & Kiekuth (1990), and Nosek & Palvia (1990).

Suggested future work
It is stated that maintenance process affects software system lifetime. Foster strongly encourages to examine the *age distribution* of software systems and acquiring other *software demography data.*

5.3.3 Tamai & Torimitsu (1992)

The article discusses software's lifetime over (product) generations. The authors focus their research on collecting statistics of *software lifetime,* investigating the state of the practice of *software reconstruction* and analyzing the factors determining software life- time. Data is being collected in two surveys.

The first, preliminary survey was performed in one company with questionnaires. The surveyed systems were written in COBOL with 30 KLOC average software size. The researchers found 32 cases of software replacements from 27 systems. Software lifetime caused by replacement varied from 2 to 20 years, average lifetime was about 9 years. Researches listed typical factors which cause replacement: *hardware replacement*, *high maintenance cost*, *change of system architecture*, *business procedures* and *social systems.*

The second survey's questionnaires were sent to 150 Japanese organizations and 42 were returned with data of organizations' software replacement cases within five years. Results from the research are listed below:

- Average software lifetime is about 10 years.
- Variance of lifetime data is large.
- Small-scale software tends to have shorter life.
- Administrative systems have longer lifetime than systems supporting business more directly.
- Some of the companies set software life length at the time of release.
- Software size grew in replacement.
- There is diversification in programming languages, but COBOL is still dominant in business applications.
- Factors that cause replacement are composite.
- More than a half of replacement cases, satisfying user requirements is given as one of the causes.
- Software, which has been replaced for reason of maintainability, has longer lifetime.

The authors point out that analysis of software lifetime gives a good support for maintenance strategy decisions. Software with short life expectation may be maintained with *quick-fix strategy*, but they should, however, be constructed by developing reusable components. Application area, programming languages, and user requirements also have effect on software lifetime.

Applicability of the approach

The article present factors, which has effect on software lifetime. A model including list of replacement factors were estimated to be better than a mechanical statistical model, since future technology development and user requirements are difficult to predict.

Constraints of the approach

No specified constraints.

Validation of the results

Preliminary survey was accomplished before the research. Data was analyzed from a relatively large set of organizations.

5.3.4 Chan et al. (1996)

This article presents a normative formalized model of software *maintenance* and *replacement effort* over a finite *fixed planning horizon*. Model extends and specifies the model of Gode *et al.* (1990) and earlier work; Chan *et al.* (1994).

If rewriting speed linear to effort is assumed, optimal timings can be given in closed-form. The represented major insights are the following:

- *Inferior current platform* implies *earlier replacement* and *compressed rewriting schedule*.
- Maintenance staff not being *familiar* with the existing system implies earlier replacement.
- Greater *functional complexity*, good *rewriting effectiveness* or poor *maintenance quality* imply compressed rewriting schedule.
- With higher rate of *maintenance requests*, rewrite appears later and replacement earlier.
- Better initial *quality* or poor rewriting effectiveness imply more relaxed rewriting schedule.
- *Potential savings from rewriting* don't come from one single feature alone but from better platform, quality and familiarity and stringent *maintenance procedure* together.

Managerial implications drawn from the previous insights are the following:

- Avoid complete rewrite of large application.
- Organize programming staff by application (to increase familiarity).
- Compress the rewriting schedule (to minimize double maintenance).
- Impose strict quality control to maintenance (to achieve low *quality deterioration rate*).

Functions and parameters of the proposed model can be interpreted as the set of factors relevant to optimal replacement timing (and resourcing). Model has the following (free) variables (referred as V1-V3)

- The *time when rewriting starts.*
- The *time when the existing software system is replaced*.
- The *size of the rewriting team*.

Model functions (referred as F1-F7)

- The *speed of the rewriting team* (function of team size, V3).
- *Functional complexity* of the existing software system at time *t*.
- *Functional complexity* of the new software system at time *t*.
- *Code quality* of the existing software system at time *t*.
- *Code quality* of the new software system at time *t*.
- *Maintenance productivity* on the existing software system at time *t* (function of complexity F2 and quality F4).
- *Maintenance productivity* on the new software system at time *t* (function of complexity F3 and quality F5).

Model parameters are the following:

- *Effort* required to develop a function point equivalent of code with the *existing* technology platform; it reflects the structuredness of the existing technology platform.
- *Effort* required to develop a function point equivalent of code with the *new* technology platform; it reflects the structuredness of the new technology platform.
- *Marginal effort* required to deal with the *functional complexity* of the *existing* software system; it reflects staff familiarity with the existing software system.
- *Marginal effort* required to deal with the *deteriorating code quality* of the *existing* software system; it reflects staff familiarity with the existing software system.
- *Marginal effort* required to deal with the *functional complexity* of the *new* software system; it reflects staff familiarity with the new software system.
- *Marginal effort* required to deal with the *deteriorating code quality* of the *new* software system; it reflects staff familiarity with the new software system.
- *Code quality* of the *existing* software system when it became operational; it reflects the control imposed on code quality during the development of the existing software system.
- *Code quality* of the *new* software system when it becomes operational; it reflects the control imposed on code quality during the development of the new software system.
- *Deterioration rate of code quality* of the *existing* software system; it reflects the control imposed on code quality during the maintenance of the existing software system.
- *Deterioration rate of code quality* of the *new* software system; it reflects the control imposed on code quality during the maintenance of the new software system.
- *Functional complexity* of the *existing* software system when it became operational; it reflects the complexity of the functional domain of the software system.
- *Average complexity of each maintenance request*.
- *Average rate of arrival of requests*; it reflects the volatility of the business environment.

Applicability of the approach

The presented model is one step closer to real life as compared to the one presented by Gode *et al.* (1990). Even though the model is rather simplistic, all the parameters are now measurable in the real world and optimal (in the sense of this model; see constraints) timing and *replacement policies* can be determined (while required parameters are given).

Constraints of the approach

Required parameters for existing and new system are the following:

- *Development and maintenance efficiency* (person hours/FP),
- *Marginal effort* required to deal with the complexity of system (person hour/FP) and deteriorating code quality (person-hours),
- *Initial code quality* (no dim.) and *complexity* (FP),
- *Maintenance quality* (1/request),
- *Average complexity of maintenance request* (FP), and
- *Rate of arrival of requests* (request/month).

Even if this input parameter data set is available, the model is (still) limited in finite fixed planning horizon and leaves out a number of cost drivers (*refactoring, changing regulations etc.*) which have a major effect on the process as a whole.

It's rather evident that this model doesn't take into account all factors necessary to determine the optimal replacement policy. And there's also this finite planning horizon which feels like a very unrealistic approach.

Validation of the results
The results deduced from the model have not been validated, but model parameters and assumptions come from field data. Constant maintenance request arrival rate is supported by field data of 10 applications over a 7 year period. Assumptions concerning system growth (Lehman's laws), maintainability (Gibson, Senn, Jones, Kafura, Reddy), complexity increase (Swanson, Beath) and maintenance productivity (Chan, Ho) are taken from results of separate research projects.

Suggested future work
1) Determination of the effect of *maintenance backlogs* is suggested. Usually some maintenance requests must be postponed (forever) to satisfy more urgent ones. When system is rewritten this backlog can be incorporated into new system.

2) Collection of data on *lifetime drivers* such as *user environment*, *effectiveness of rewriting*, *technology platform*, *development quality*, *software familiarity* and *maintenance quality* of the existing and the new software systems.

Software lifetime drivers
See model functions and model parameters abowe.

5.3.5 Sahin & Zahedi (2001)

The article presents a model for *strategic decision making* in software maintenance. The model is described at general level. Accurate formulas are presented in authors' other article.

The model's horizon axis is *upgrade cycle* of the software product and vertical axis is *customer satisfaction index*. Customer satisfaction can be classified into eight *policy baselines* based on three categories of change actions: *warranty*, *maintenance* and *upgrade*. Each baseline has a recommendation for needed action to increase the satisfaction or to keep it at the same level.

The authors found that *high quality* is a very relevant variable and with higher *quality of design* and *implementation*, higher *average returns* are achieved. Furthermore, if software product is of high quality, customer's reactions have less severe influence on average return.

The results indicate that average return is higher under *high volatility markets* than under less volatile markets due to the larger possibilities to upgrade the system. But on the other hand, in

high volatility markets, quality and *technological obsolescence* have more important roles to customers' satisfaction and average return.

Applicability of the approach
The model presented in the article is relevant to ELTIS. It can be used to analyze organizations' portfolio of software systems and to give support in decision making of needed maintenance actions.

Constraints of the approach
Details of the model are presented in another article.

Validation of the results
Model is validated with 3,840 base scenarios and 23,040 variations of scenarios in 487 organizations.

5.4 Function point -based estimation

5.4.1 Furey (1997)

In this short article the author argues that *function points* (FP) should be used as a size and complexity measure. He prefers function points, because they are independent of development tool and technology, consistent and repeatable, thanks to defined and documented process, and they help normalize data. In addition, function points enable comparisons between different technologies. They can be counted in an early phase and used in *effort*, *schedule* and *defect estimation*.

Applicability of the approach
This article lists arguments, why function points should be used instead of LOC in software size estimations.

Constraints of the approach
No explicated constraints.

Validation of the results
The article doesn't present any kind of testing or comparing of FP with LOC.

5.4.2 Kitchenham (1997)

In this short article the author states that specific types of *function points* (FP, Albrecht's and Symons Mark 2 versions) aren't reliable, since they are not straightforward and simple measures and have fundamental flaws in their construction. In addition, counter judgements have an effect in function points counting. However, the author doesn't prefer LOC as a size

measure. She proposes that function points should be further improved and used by taking into account their limitations.

Applicability and constraints of the approach
The article lists arguments, why function points should be used only with caution. Constraints have to be taken into account also in our research (in case that function points would be used as software size measure).

Validation of the results
The article doesn't present any kind of testing or comparing of FP with LOC.

5.4.3 Abran et al. (2002)

This research apply *functional size measurement* in building estimation models for software maintenance. Authors describe two separate software maintenance research projects. Development of enhancive maintenance effort estimation models using functional size was studied. Following two hypothesis were examined: 1) *maintenance effort* is increasing relative to *functional size*, 2) if correlation between maintenance effort and functional size is not significant, there exists other factors which together with functional size have influence on required maintenance effort.

Functional sizes were measured using version 2.0 of second generation function point method called *COSMIC-FFP*. Data from field research A (a web-based system) was best explained with *functional size*, *project difficulty* and *experience of maintenance personnel* as variables (explanatory power, $R^2 = 0.83$ and $R^2 = 0.57$ in separate groups). Data from field research B (a real-time system) was best explained with 2 variable regression model $y = ax + bz + cxz + d$ (x = *size*, y = *difficulty*). Using that model coefficient of determination rose to significant level $R^2 = 0.84$. Functional size alone doesn't explain maintenance effort. However, together with project difficulty it does. Second hypothesis was supported by these research projects.

Applicability of the approach
This research shows that there exists domains where accurate early maintenance effort estimation is possible. It would be surprising if this would appear to be possible only in the domains examined here. However, due to the use of regression models, no extra information (supporting decision making, references to legacy data, *etc*.) besides the effort estimate value is provided.

Constraints of the approach
These results are domain specific and shouldn't be wildly generalized.

Validation of the results

There were 36 maintenance projects, 21 of which were maintenance of large real-time system (field research B) and 15 maintenance of web-based linguistic system (field research A).

Suggested future work
Better defined metrics for project difficulty are needed to refine results. More maintenance data from different domains of business is needed to further test the second hypothesis.

5.4.4 Other works on the use of function points in software maintenance

Other works include Abran & Robillard (1993); Engelhart (1995); Tran-Cao *et al.* (2002).

5.5 Dynamic maintenance effort estimation

5.5.1 Jørgensen (1995)

This article describes experiences from the development and use of 11 software maintenance effort prediction models falling into 3 base categories: regression analysis, neural networks and pattern recognition.

Best models achieved 50% MdMRE (*Median Magnitude of Relative Error*). Expert predictions ranged from 10% to 20% on the same scale. However, there's a major difference between environments. While models were compared to historical data, the managers (most likely) had their data from an environment where the maintainers knew about the predictions (interpreted by the maintainers as "plans") in advance!

Jørgensen suggests that prediction models should be used as instruments to support the expert estimates and to analyze the impact of the maintenance variables on the maintenance process and product. Pattern recognition approach seems to have the potential for being superior to the other approaches in supporting the expert. Pattern recognition models (described *e.g.* in Briand *et al.*, 1992) are able to point out a set of similar maintenance tasks to the one to be predicted.

Applicability of the approach
Results are important to us. If we are aiming at supporting decision making, we need to have something more than a black box competing with the expert. Pattern recognition or hybrid approach is capable of producing supporting information in addition to the estimated maintenance effort.

Constraints of the approach
Not explicated.

Validation of the results
Data was collected from 109 randomly selected maintenance tasks executed by 110 maintainers of 70 applications written in COBOL or some 4GL, ranging 5-500 KLOC in size and 1-20 years in age.

Suggested future work
Comparison of the prediction accuracy of expert predictions and the prediction accuracy of formal prediction models, when both types of predictions are carried out under the same conditions.

5.5.2 Caivano et al. (2001)

This article presents a *dynamic maintenance effort estimation model* and supporting tool (*DEE*). The model expects process performance to change during project and tries to adapt itself to reflect the changes. First estimator is deduced from previous projects pretty much the same way as in other estimation models (regression analysis). That estimation is then further refined during project execution using newly collected metrics.

DEE is built on Access 2000® and StatSoft STATISTICA®. External inputs come from past experience and current project. Used econometric model is built automatically using (forward stepwise multiple) regression analysis. Model refinement is done every time the accuracy of estimation drops below given threshold value.

There were the following lessons which were learned on *system renewal process*.
- A renewal project can be more efficient if supported by tested tools.
- A restoration process (a process resembling reverse engineering and reengineering) is useful when intention is to improve quality without altering software structure.
- Expectations for some variables (*e.g.* complexity gain) can be limited to reduce required effort.

These “lessons” seem quite self evident and the connection between them and the subject of research was left unclear.

Applicability of the approach
The results are not directly applicable to ELTIS. The tool (implementing the presented model) is used when refining effort forecast during rejuvenation (reengineering) process. At that point the decision has already been made to modernize the old system.

Constraints of the approach
Model is defined only by the tool (DEE) implementing it, thus the tool is a requirement.

Validation of the results

There are no convincing validation. There exists only an experimental test with one renewal project of aged banking application and some simulations with legacy data.

5.5.3 Other related works

These include Calzolari *et al.* (1998).

5.6 General maintenance cost drivers

5.6.1 Niessink & van Vliet (1998)

The article presents results from two measurements of cost drivers of software maintenance. Research included two organizations: the first IT department is responsible for Dutch social security system (A) and the second is a part of the Dutch industrial organization (B). The selection of possible drivers was based on literature and interviews of managers and engineers. To analyze the data, researchers used principal component analysis and multiple regression analysis.

Although the environments and measurement programs were quite similar in organizations, data from organization A was considered more useful to explain a variance in effort to implement change request than organization B's data. The authors point out the implication of a *consistent use of standardized measurement process*. To improve the overall prediction, authors suggest looking for variables relevant to analysis and testing.

There were a versatile set of data collected (of which many potential maintenance cost drivers). These included the following: *maintenance type*, *software complexity, requirement changes, size, fault correction effort* (*fault locality, cumulative changes made to the software, characteristics of the defective software components*), *work needed to convert data, changed use of database*, *user interface change*, *code attributes* (*structuredness*, *readability* and *quality*), *experience* (of the engineer with the code), *kind of database used*, *relationship with other applications*, *relationship with other change requests, documentation* (*readability, completeness*, *clarity* and *structure*), *availability of test sets, tests performed*, *complexity* and *size of the change*, *size of the code to be changed* and other application characteristics.

Applicability of the approach
The results are restricted to the studied organizations and more research is required to determine the presented formulas for maintenance effort more accurately. As noted earlier, the article includes a wide list of attributes possibly having effect on maintenance costs.

Constraints of the approach
As noted in the previous paragraph.

Validation of the results
Data was collected from two organizations but results aren't generalizable.

Other related works
Niessink & van Vliet (1997).

5.6.2 Jørgensen & Sjøberg (2002)

The research focuses on *experience's* effect on *maintenance skills*. The skills were measured as a frequency of *unexpected major problems* and as accuracy of *prediction of maintenance problems.* Also maintainers' learning from experience was studied.

The research was accomplished in a software maintenance department of a Norwegian company. The department maintained over 70 applications, mainly written in COBOL, 4GLs or C. The ages of the software varied from less than a year to more than 20 years and sizes varied from a few thousand to 500,000 lines of code. From 110 workers, 54 maintainers were randomly selected for interviews. The maintainers' average experience in maintaining and/or developing was 7.7 years, from which 3.4 on average were spent in maintaining the certain software.

The major findings were the following:
- Experience decreases the frequency of unexpected major problems to a certain skill level.
- The maintainers' *general experience* and *application specific experience* didn't have a great effect on maintenance problem prediction accuracy.
- A simple one-variable model produces more accurate predictions than maintainers.

Following data-attributes were collected during the research: *total maintenance experience*, *maintenance experience on the application* to be maintained, *application development experience*, *task solving confidence*, *major unexpected problems*, *prediction accuracy* and *size of the task*.

Applicability of the approach
According to the article, it is important to consider maintainers' skills as an attribute to maintenance productivity (which affects required maintenance effort and thus profitability of extended system maintenance and modernization).

Constraints of the approach
No specific constraints.

Validation of the results
As noted earlier, empirical data was collected. Findings are similar to previous studies. The author lists the following possible effects on the validity of the results:

- Bias in the allocation of maintenance tasks; difficult tasks were given to most experienced maintainers.
- Lack of realism of prediction process; interviewer's identity (researcher versus project leader) may have an effect on answers.
- Low quality of the data; there is a risk for misunderstanding a question in interviews.
- Lack of meaningful measures; it is difficult to measure experience.

Suggested further research
At the end of the article, the authors state their interest to extend the research by focusing on learning and training processes' effect on maintenance and prediction skills.

5.6.3 Other works

Other works in the area include Gorla & Benander (1990); Mancini & Ciampoli (1990); Gerlich & Denskat (1994); Bredero *et al.* (1995); Hürten R. *et al.* (1996).

5.7 Software complexity effects

5.7.1 Gibson et al. (1989)

The authors of this article have examined the *system structure*'s impact on software *maintenance performance*. Motivation for the research is a fact, that most of the maintainers' time is spent on understanding the system to be maintained. There is also empirical evidence, that complex programs need more maintenance than less complex ones.

Three professional programmers were assigned to do three maintenance tasks to three different versions of a system written in COBOL. Six metrics (*Halstead's E, McCabe's V(G), Woodward's K, Gaffney's Jumps, Chen's MIN* and *Benyon_tinker's C2*) were tested.

Results indicate that improvements in system structure decreased total maintenance time and error frequency. This applies only across a portfolio of tasks, not in specific task. Programmers were not found to be aware of structural differences although differences improved performance. Also metrics were considered as a potential tool for project management. However, programmers couldn't separate complexity of the system from complexity of the maintenance task and therefore systems were not ranked inconsistently by them.

Following attributes were used in the research: participants' background (*age, titles, general/development/maintenance experience*, and COBOL/ISAM experience), *programmer's performance* (*maintenance time, accuracy of modification, confidence* and *perceptions*) and *maintenance task's difficulty* and *complexity metrics* (as described earlier).

Applicability of the approach
Also in our research structure of the system must be considered. The article gives a starting point by representing empirical analysis of the effect of system structure to maintenance, but reported research offers only thin research data and has some flaws, for example, in estimation of complexity (in methods).

Constraints of the approach
The results aren't directly generalizable.

Validation of the results
Only three versions of a single program were investigated.

Suggested future work
Future research is stated to be needed related to the dimensions of programmer perceptions of complexity and determination whether the received results of relationship between system structure and maintenance performance exist in "real-world" settings.

5.7.2 Banker et al. (1993)

The article analyzes the relation between software complexity and maintenance costs by further developing an economic model of software maintenance presented in an earlier article of Banker *et al.* (1991). Previous works on the topic are also summarized.

Complexity's effect on maintenance costs was measured through software comprehension and project factors (*e.g.*, *expended hours*, *software size*). The researchers found that complexity has a significant impact on maintenance costs.

Following maintenance cost drivers can be identified:
- *Maintainer skill*,
- *Maintainer application experience*,
- *Structured analysis/design methodology used*,
- *Operational quality*,
- *Hardware response time*, and
- *Complexity* (measured in three dimensions: *module size*, *procedure size*, and *branching complexity*).

Size was measured with *source lines of code* and *function points*, and effort with *hours charged to the project*. Modularity was mentioned as a measure of complexity.

Applicability of the approach

The model is described accurately and results can be used as a guideline for future research. Researchers stated, that results of the research are valid only in case of the specified (large) set of commercial IS applications. Other sites need future research.

Constraints of the approach
Values of parameters (*e.g.*, optimal module size) differ based on used programming languages.

Validation of the approach
The research included 65 maintenance projects from 17 applications from a large commercial bank. The applications were written in COBOL.

5.7.3 Kemerer (1995)

The article presents a review of empirical research literature focused on a relationship between *software complexity* and *software maintenance performance*. The survey included 61 articles, which the author had briefly summarized and classified into categories.

The first category includes articles discussing *modularity* and *structure metrics*. Researches have find out, that larger modules have fewer errors than smaller ones and optimal module size is not too small nor too large. In addition, *global variables* and *high degree of coupling* cause more source code modifications and increase error rate.

The complexity metrics section presents, that *SLOC* is a reliable metric for measuring complexity. The last section; comprehension research, lists main problems in comprehension: *personnel turnover*, *difficulty in understanding the program* and *difficulty in determining impact domain*. According to research results, *experience* (in years), *breadth of experience*, *knowledge of the system* and *efficiency of the aids* have a positive effect on maintenance.

Applicability of the approach
Since the article summarizes main results from previous studies it is a good reference to a literature in software maintenance complexity research.

Constraints of the approach
No specified constraints.

Validation of the results
The article summarizes previous research results.

Suggested further research
The author points out, that software maintenance has been understudied relative to its practical importance.

5.7.4 Munson & Elbaum (1998)

Complexity metrics can provide valuable information on software modules to be used during testing. This study presents *metrics* depicting how each software system *revision* differs from it's successor and predecessor in terms of faults.

Relative complexity is a weighted sum of basic (raw) source code metrics of single release. *Code delta* and *code churn* are derived from relative complexities of several releases. This study proves that relative complexity together with code delta and code churn are closely related to code quality. Code churn and *rate of trouble reports* had (Pearson) correlation of 0.65 (significant correlation). Relative complexity provides information on fault injection process (which directly correlates to change in relative complexity).

Applicability of the approach
The study shows that complexity metrics really *can* provide information on *fault proneness* and presumable *fault insertion rate*. This information is vital when trying to predict the development trend of maintenance effort.

Constraints of the approach
Metrics are not specifically defined in this paper. The research team of Munson & Elbaum developed their own supporting software including raw metrics analyzer (*CMA*, C Metric Analyzer), relative complexity calculation (*RCM*, Relative Complexity Metric) and code churn calculation (*EVOLV*).

Validation of the results
A large embedded real-time system (300 KLOC, 3,700 modules, programmed in C) was evaluated over 19 successive versions.

5.7.5 Polo et al. (2001)

This paper presents an empirical study on the correlation of simple code metrics (*LOC*, *module count per application*) and *maintenance effort*. It aims (but only partly succeeds) to provide a method for the estimation of maintenance in the initial stages of *outsourcing maintenance projects*, when there is very little information available on the software to be maintained.

Logistic regression analysis is used to derive the model. Resulting prediction equations (functions of metrics) for corrective maintenance allow to categorize applications to "problematic" and "non-problematic" (from the Service Level Agreements point of view). Other (negative) results confirm the results from other researchers stating that size is not a good predictor of *fault-proneness*.

Applicability of the approach
Taken other results in account, it does not seem very likely that this approach (only measuring size) would appear to be useful. Defects requiring corrective maintenance are focused on few modules and the need of perfective maintenance does not heavily depend on the size of the software system.

Constraints of the approach
No explicate constraints.

Validation of the results
Validation data was collected from two sets of banking applications over maintenance period of two years. Both sets of applications were developed in COBOL/CICS on top of DB2. They consisted of several MLOC and they are maintained by Atos ODS.

Suggested future work
Authors present correlation between code metrics and maintenance effort as future work.

5.7.6 De Lucia et al. (2002)

This paper presents a model for an early maintenance effort estimation. Metrics were chosen using correlation analysis. The model was built using regression analysis and validated against massive adaptive maintenance process used by EDS SC.

The used metrics were the following:
- *Number of software code components* in the work-packet (incremental part of software in massive maintenance process),
- *LOC*,
- *McCabe cyclomatic complexity,*
- *Number of control variables,*
- *Halstead software science volume,*
- *Number of logical branches not used*, and
- *Actual effort of the work-packet* (measured in man-days).

Applicability of the approach
This estimation method is (in principle) directly applicable, at least on Y2K or euro-conversion – and like massive maintenance projects. In addition, it is likely that required parameters are early (*i.e.* before the project is started) and easily available, which are important features when trying to support decision making. Due the nature of the model, it's a *black-box model*, and doesn't provide any additional information besides the effort estimation.

Constraints of the approach

The application of the model requires a sample set of maintenance projects and needed metrics from them, such as: number of programs, one dimensional metric, and one structural metric, LOC and cyclomatic complexity (which were used in this paper).

Validation of the results
The presented model was validated against data collected from a large Y2K remediation project. There were 40 KLOC of components, and 15 KLOC of them were modified. The programs were written in COBOL/CICS, PL/1, JCL, and Assembler. A leave-one-out cross-validation was performed. Average prediction error was 47% with 10% sample, 42% with 30% sample and 35% with full data. Thus the model works pretty well with relatively small sample sets.

Other related works
Other work by the authors is De Lucia *et al.* (2001).

5.7.7 Other works

Yet another relatively recent experience report on collecting maintenance metrics data in case of industry-software is (Fasolino *et al.*, 2000).

5.8 Maintainability

5.8.1 Coleman et al. (1994)

This article represents a comparison of metrics based *maintainability evaluation models*. These evaluation models can be used to determine when a system should be *reengineered*. There exists the following five models for this purpose: hierarchical multidimensional model, polynomial regression model, aggregate complexity model, principal component analysis, and factor analysis.

By expert review, two (hierarchical, and polynomial) of these five models were selected to be actually applied to industrial software systems. Both of them produced results corresponding to the maintenance engineers' intuition and also provided useful additional data. Aim was to develop a simple maintenance assessment mechanism for "line" engineers to use in ensuring that system maintainability doesn't decline on modifications.

Applicability of the approach
Metrics-based model of system maintainability evaluation is simple and based solely on *source code metrics*. It could be of use when information on general profiling of system maintainability or especially on guiding how to focus refactoring is required. However, weak validation suggests that this model should be applied with caution.

Constraints of the approach

Tools to obtain the required metrics are needed.

Validation of the results
Two C/Unix systems (240 KLOC, and 3 KLOC) were used as test cases of the model. Expert judgment on the maintainability of system A was "low" and of system B "high". Maintainability index given by the polynomial regression model suggested that 33.4% of the code of system A had low maintainability and only 2.8% of system B. Maintainability indices obtained by the models were not validated against actual maintenance costs/effort required or against wider expert judgments on maintainability.

5.8.2 Lanning & Khoshgoftaar (1994)

The article discusses *code complexity*'s effect in *maintenance difficulty*. The relationship between these two factors can't be measured directly and authors apply canonical correlation analysis in their investigation. The method is applied in the system test phase of a commercial real-time product. The product consists of about 223,000 lines of Assembly code in 152 files. Product's main purpose is to provide stable interface for software products, which are written to a varying hardware base.

Following attributes were used in the research: *complexity* (*number of unique operators and operands, total number of operators and operands, number of executable statements, McCabe's cyclomatic number, number of times the control flow crosses itself, number of calls out and calls in, average information content classification*) and *maintenance difficulty* (*added/deleted/moved, noncomment source lines, number of program faults,* and *number of design changes*).

Applicability of the approach
The method is presented exactly, but isn't reliable without further development, because model omits influences on maintenance difficulty.

Constraints of the approach
Results can't be used in general, because the research includes only one project.

Validation of the results
Researchers investigated system test phase of a one commercial real-time product. The authors note, that results are restricted to their investigations, because canonical correlation analysis is presented as a restricted form and the model omitted some influences on maintenance difficulty. However, they considered canonical correlation analysis as a useful exploratory tool.

Suggested further work
Authors' future research will focus on developing general soft models of the software development process for both exploratory analysis and prediction of future performance.

5.8.3 Pearse & Oman (1995)

The article presents a research, in which maintainability metrics are used to measure the effect of adaptive and perfective maintenance. There are three aspects in maintenance which are discussed here: *management practices*, *operational environment* and *target software system*. The authors restrict their investigation into source code of the target software system.

The researcher constructed approximately 50 regression models. The best of these models was based on *Halstead's Volume*, the extended version of *McCabe's cyclomatic complexity, lines of code* and *percent of comment lines*. These four metrics were used in calculation of *Maintainability Index* (MI). Researches selected four maintenance activities: unused code removal, compiler warning removal, code restructuring and integrating new features. Pre-post analysis was used to measure maintainability before and after maintenance task.

Good results were achieved in testing of MI-model. However, the researcher noticed that it doesn't reflect to any kind of maintenance tasks, for example unused code removal and compiler warning removal. These results suggest that assessment tools should also provide data with MI for interpreting the value.

Applicability of the approach
MI is a simple model, and therefore it misses detailed information. Thus it is not useful to us.

Constraints of the approach
Two tools: UX-metrics and Micalc, were used in research to calculate the MI-value.

Validation of the results
The models were constructed in cooperation with Hewlett-Packard.

Other related works
Oman & Hagemeister (1994) and Oman & Hagemeister (1992) in which is provided a classification of target system metrics.

5.8.4 Other related works

These include: Rose & Eriksson (1998); Sheldon *et al.* (2002).

5.9 Project size effects

5.9.1 Banker & Slaughter (1994)

The article focuses on the relationship between *maintenance project size* and *productivity.* Project size's influence on productivity was analyzed by using Data Envelopment Analysis (DEA) and DEA-based heuristics were used to examine returns to productivity scale.

It was found that project size's effect to productivity should be an important measure in maintenance and development projects. Smaller maintenance actions should be grouped into larger releases. Presence or absence of scale economies at given maintenance project size has influence on maintenance productivity. Researchers also noticed, that the most productive size for the project is larger than 90% of the projects within the sample of this study. Identifiable maintenance cost drivers included *work efficiency* (measured with hours and function points).

Applicability of the approach
The research focused in project size's effect on maintenance productivity. Results are useful to project managers. In our point of view, using DEA-based heuristics to examine returns to scale for the projects is possibly interesting.

Constraints of the approach
No explicated constraints.

Validation of the results
Data was collected from 27 software maintenance projects from a major mass merchandising retailer in two years period. Programs were written in COBOL. Research results were also tested with heuristics and analyses. Test results appear to confirm the robustness of research results.

Suggested future work
Authors mentioned a few possible extensions to research:

- Determining whether the received findings can be replicated in case of other software maintenance projects.
- Identifying factors contributing organizations' ability to manage maintenance projects.
- Identifying other factors' effects on maintenance productivity.
- Identifying DEA methodology's usability to assess the performance of an organization which utilizes a proactive change management program.

5.10 Other potential maintenance cost drivers or metrics

There are also at least the following classes of decision criteria to be considered while making software modernization decisions. Some of these factors have already been mentioned above related to the individual studies. Since an incomplete model may give misleading results, it

should be noted that the models should be iteratively enhanced, calibrated and validated within the target-organizations. If a quantitative model would be a goal, it should be ensured that sufficiently complete metrics data is available.

For receiving sufficient explanatory power for the decision model, the key elements characterizing the maintenance situation should be identified. These additional factors include characteristics describing the organization, project and system. Time aspect should also be noted. It would be good if reliable and relevant statistical or metrics data would be available, although the reliability of many of the quantitative basic metrics, such as LOC and cyclomatic complexity is questionable.

5.10.1 Factors related to regulators

- Jurisdiction and other regulations, which pose more or less rigid boundary conditions to the acceptable solutions.

5.10.2 Factors related to software business processes

- Real needs of customers (customer-driven development) vs assumed needs.
- Really useful technological opportunities.
- "Hype" (*i.e.* needs and expectations created merely by unfounded promises and wishful thinking) (including merely technology-driven development).

5.10.3 Technical factors

- Technical options (possibilities) and their technical quality metrics.
- Maintainability (*pre* & *post* modernization) and costs due to maintenance.
- Lifetime of technologies (some possibly becoming *obsolete*).
- Possibilities to change the system sufficiently quickly to meet changing customer requirements (*version cycles*).
- Effects of new adopted/to be adopted *modernization techniques* and support technologies (such as reverse engineering) *etc*.

5.10.4 The general type of the software and the applications area

- Type of software (administrative, embedded, real-time system etc.).
- *Novelty* of the application area (*e.g.* standard invoicing vs. newest mobile technology).
- Effects of changes in environment (in addition to above-mentioned regulator-based changes).
- General requirements (*efficiency-*, *timing-*, *memory* restrictions (*RAM*, *disk-space*)).

5.10.5 User requirements

- Typical, average amount of *source lines changed* (*added*, *deleted* or *modified*)/total size of the system/time unit.
- Requirements volatility (look: Stark *et al.*, 1999; Di Lucca *et al.*, 2002).

5.10.6 Quality of available human-resources

- *Availability* of the original coders (as councellors for new ones).
- Relevant technical *work experience*.
- Relevant work experience on the application domain.
- General maintenance experience.
- Maintenance experience of the system to be maintained.
- Work efficiency.

5.10.7 Applied maintenance process models

- Systematic configuration management (*e.g.* Capretz & Munro, 1994).
- Quality/sufficiency of the used CM-tools.
- Systematic *process models* of error-corrections (Jambor-Sadeghi *et al.*, 1994; Eisenstadt, 1997; Kajko-Matsson, 2002; Agans, 2002).
- Collected *feedback* from the users (error-reports *etc.*), and related process improvement.

5.10.8 Specific properties of the software

- *Size* of the system (*KLOC*, *KDSI*, *number of tokens*).
- *Age* of the system.
- *Software complexity* (*e.g.* average module size, *number of modules*, *McCabe's cyclomatic complexity*, *control complexity*, *Halstead's data complexity*, *code redundancy* (Burd & Munro, 1997), *metrics values of object hierarchy*; Kiran *et al.* (1997). Software complexity is one of the main maintenance cost-drivers (Banker *et al.*, 1991; 1993). The reliability of complexity data is enhanced if versatile metrics data is collected (Kafura & Reddy, 1987), although its collection may be laborious, and thus potentially not cost-effective.
- (Un)*structuredness* (*e.g.* gotos).
- *Delocalization of system logic* (*delocalized program plans*; Letovsky & Soloway, 1986).
- (Module) *cohesion*.
- (Module) *coupling*.
- Number of *versions*.
- Number of *variants*.
- Number of *releases*.

- Level of using established *standards* (*e.g.* programming languages, graphical interfaces, databases).
- Reuse (components acquired from elsewhere/those developed within the organization).
- Program comprehensibility, affected by numerous factors, *e.g. naming conventions* of symbols (Laitinen, 1995), *length of identifiers*.

5.10.9 Used basic programming tools

- Programming language(s) (and its abstraction level: Assembler,... C,..., C++,... application generators).
- Used compilers and debuggers.

5.10.10 Documentation

- *Amount of documentation* (pages).
- *Quality of documentation* (*completeness*, *accuracy*, *timeliness*, *compactness*, *comprehensibility*, *readability*, *consistency*, *history information* about system development and errors).
- *Amount of comments* (and its *adequacy*).
- *Quality of comments* (focus: functions, modules, data-structures, definitions, complex structures *etc.*; Riecken *et al.*, 1991).
- *Programming style* (applied standards, coherence; Oman & Cook, 1990; 1991).
- *Traceability of design decisions*.

5.10.11 Design of the system

- Applied *quality assurance* techniques (*e.g.* code inspections, testing procedures).
- General quality (metrics: *e.g. mean time to failure*; MTTF; *rate of failure* (ROCOF); *probability of failure on demand*; POFOD; *usability*, *mean time to change*; MTTC, error correction costs after system delivery; *i.e. spoilage*, *security*; *robustness*, *integrity*).
- Applied design principles (*e.g. logicality*, *structuredness*, *modularity*, *object-orientation*, *information hiding*, *speculative design*, *flexibility*).

5.10.12 Factors affecting maintainability

- *Expected lifetime* (neglected maintainability reduces this, due to *e.g.* prototyping, optimizations).
- *Process metrics* (*number of corrective maintenance requests*, *average time used to impact analysis*, *number of considerable change-requests*, *amount of user-interaction*; Sommerville, 1996).
- *Generality* (preparedness to operation on *e.g.*: different hardware, operating systems, input/output formats, data-structures, algorithms, and *portability*).

5.10.13 Applied solutions supporting maintenance

The effects of applying sophisticated maintenance support techniques should be taken into account while estimating the effort that maintenance requires. These techniques are listed in the following chapter.

6 APPROACHES FOR SOFTWARE MODERNIZATION AND ITS SUPPORT

In this chapter we compactly list the main available branches of solutions for software modernization and its support. Since the project objectives in the focus area of this chapter are yet not sufficiently specified, we will here not delve into the details of the many branches of potential solutions.The purpose is to provide initial framework of the factors affecting the success of software modernization. Reengineering and reverse engineering are the most important sub-categories. Two last categories relate to the long-term, proactive maintainability enhancements.

6.1 General organizational decisions

Managers may affect the following aspects, which provide frames for software maintenance.

- Principles of selecting maintenance personnel (*per* system).
- Training, compensations, incentives, status (Landsbaum, 1992).
- Maintenance organization (Swanson & Beath, 1990; Yeh & Jeng, 2002).
- Decisions regarding *e.g.* tools to be used, standards to be followed and attitude towards reuse, code change practices, and applied process models.

6.2 Configuration management

Systematic configuration management (Berlack, 1991; Capretz & Munro, 1994; Tichy, 1995; Lyon, 1999; Leon, 2000; Haug *et al.*, 2001) is a necessity while maintaining large systems and its importance is emphasized at the latter phases of system's life-cycle. In practice, configuration management is done by CM-tools (*e.g.* RCS, SCCS, make, ClearCASE, CVS), which typically support control of code change rights, determination of base lines, automatic change reporting, change accept control, and determination of program deltas.

6.3 Re-engineering

Re-engineering means studying the software system and changing it into (often also) functionally new form. Main text-books written in English are the following: Arnold, 1993; Miller, 1998; Warren, 1999; Ulrich, 2002; Valenti, 2002. Other related relatively recent works include Bray & Hess (1995); Baniassad & Murphy (1998); Fanta & Rajlich (1998; 1999); Chu *et al.* (2000); Tahvildari & Kontogiannis (2002); Zou & Kontogiannis (2002). Look also Bianchi *et al.* (2003) for data reengineering.

6.3.1 Sneed (1995b)

In the article the author presents a five-step *reengineering planning process* for estimating whether reengineering is worth the required effort. The process model includes following five steps:

1) *Portfolio justification*: calculating *return on investment* by analyzing *enhanced business value*, software's *quality increase* and improvement in maintenance process.
2) *Portfolio analysis*: applications' need for reengineering is prioritized according to their *technical quality* and *current business value*.
3) *Cost estimation*: calculating *estimated costs* by identifying and weighting the software's components.
4) *Cost-benefit analysis*: comparing the estimated costs with *benefits to be achieved* in A) reengineering, B) redeveloping and C) doing nothing at all.
5) *Contracting*: contract can be based on time and material or results.

The article mentions the following possible measurements: *lines of code*, *number of databases*, *number of files*, *number of fields*, *database accesses*, *number of function points*, *data complexity*, *cyclomatic complexity*, *interface complexity*, *data-access complexity*, *number of relationships among files*, *degree of module coupling*, *distance between variable references*, *general quality*, *data dependency rate*, *number of elementary data elements*, *modularity*, *testability*, *portability*, *test environment*, *test support*, *number of test cases*, *average cost of test case*, *productivity rate*, *annual maintenance cost* (*current cost*, *cost after reengineering* and *cost after redevelopment*), *operation cost* (specified more precisely as in case of the previous attribute), *business value* (*current business value*, *business value after reengineering* and *business value of a new system*), estimated *reengineering costs* and *redeveloping costs*, *time and risk factor*, *expected life(-time) of a system*, *user satisfaction* and *maintenance programmer morale*.

Applicability of the approach
Accurate formulas are presented and the reengineering planning process is relevant to ELTIS.

Constraints of the approach
Measurement program is needed in project justification.

Validation of the results
The presented planning process has been developed with 25 years programming experience.

Related works
A newer work on reengineering risks is (Sneed, 1999).

6.3.2 Ransom et al. (1998)

Article presents an assessment method (planning part of RENAISSANCE method) that examines a legacy system from its technical, business and organizational perspectives. The method guides users through assessment of these perspectives and provides further guidance on interpreting the results obtained from assessment. Method is iterative by nature and it supports quick rough estimates and more detailed ones by further iterations.

There are the following two base principles for the method:

- Reengineering must be company and project specific. The method is designed so that it can be instantiated according to particular company and project requirements.
- Both the reengineered process and the reengineered system must be continuously refined. The principal product of applying the method is a system transformed to evolutionary state (from whatever "legacy" state it had before).

The product of the assessment method is to gain a sufficient depth of understanding of the legacy system. Typical questions answered during assessment are

- Is the system critical to the organisation in which it operates?
- What are the organization's business goals?
- What are the evolution requirements?
- What is the anticipated lifetime of the system?
- What is the required lifetime of the system?
- What is the technical state of the system?
- Is the organization that operates the system amenable to change?
- Does the organization responsible for evolving the system have sufficient resources?

Assessment process starts with method instantiation. Next phase includes business value, external environment and application assessments, which are carried out in parallel. Finally results are interpreted to find the optimal evolution strategy.

Instantiation includes defining assessment technique (expert judgement, quantitative metrics) and level of detail. The goal of business value assessment is to determine the importance of the system to the organization.

The external (technical) environment of a system is the union of hardware, supporting software and organization's infrastructure. The following hardware characteristics are suggested to be considered: *vendor/supplier rating, maintenance costs, failure rate, age, ability* to perform function, *performance*. The following supporting software characteristics are suggested to be considered: *License costs, Frequency of fixes/patches, Quality of support personnel*. The following organizational factors are suggested to be considered: *Type of organization* and system users (how skilled is the system user's work? what is in-house and what is outsourced? *etc.*), *Technical maturity* of the organisation, *Training procedures* in the organization, *Skill levels* of system support, and *Organizational attitude* to change.

Application assessment is concerned with the application software of the legacy system in question. The following characteristics are considered: *complexity, data, documentation, external dependencies, legality, maintenance record, size, security* and *test bed*.

Applicability of the approach

The represented method framework is highly relevant to the decision making support goals of ELTIS. In addition a great deal of factors the authors believe to be relevant to the optimal software evolution strategy are listed.

Constraints of the approach
These are not explicated.

Validation of the results
"The assessment method is currently being evaluated as part of the RENAISSANCE method by industrial organizations involved in the project. We aim to incorporate their feedback in subsequent refinement of the method."

6.3.3 Teng et al. (1998)

In the article authors have collected empirical data from 105 organizations' (sizes vary from 5,000 to 10,000 employees) business process reengineering projects by comparing *projects' radicalness* and *stage-effort profile* to projects' *implementation success*. The researchers use correlation analysis in evaluation of responses.

The authors found a strong positive relationship between radicalness and implementation success. Also *roles and responsibilities*, *information technology* and *changes in the work flow patterns* have a great influence to perceived success. Additionally, attributes used in the work included also the following: *extend of change*, *strength of effort*, *goal fulfillment*, *number of employees*, *business processes*, *organization's type*, *used formal methodologies*.

Applicability of the approach
The method used in the article is useful in analyzing companies' reengineering processes.

Constraints of the approach
No constraints for using this research method.

Validation of the results
As noted empirical data was collected from a large set of organizations. In the discussion section the authors note, that the selected projects may not be a comprehensive variety of the reengineering projects. Therefore the results should be interpreted with caution.

Suggested further research
The authors emphasize, that more attention should be shifted from analyzing the existing business procedures to social design and process transformation in reengineering projects.

6.3.4 Comella-Dorda et al. (2000)

In the article system evolution is defined as a continuum of system *maintenance*, *modernization* and *replacement*. The authors describe maintenance, replacement and white-

box modernization in general and focus on *black-box modernization*, in which understanding of the system is gained by examining merely its inputs and outputs.

The authors emphasize, that all options to modernize a legacy system must be thoroughly explored and evaluated. Following techniques are discussed:

- User interface modernization
  - *Screen scraping* for user interface modernization (in which old, text-based interface is wrapped with graphical interface).
    + cost
    – maintainability
- *Data modernization* for accessing data with a different interface or protocol.
  - *Database gateway*. (wrapping legacy data with standard protocol)
    + cost
    + tool support
    – maintainability
  - *XML* integration. (convert proprietary connection between systems to XML-server based)
    + flexibility
    – evolving technology
- Functional modernization for encapsulating the data and business logic.
  - *CGI* integration. (wrap legacy data and functionality behind web-interface)
    + cost
    – flexibility
  - Object-oriented *wrapping*.
    + flexibility
    – cost
  - Component wrapping.
    + flexibility
    – cost

All the black-box techniques require the legacy system to be stable, because the new functionality completely relies upon the old. Especially screen scraping (sometimes referred as "whipped cream over road kill") is very sensitive to the underlying application.

Object wrapping and componentization come in handy when the legacy system is to be incrementally replaced. Modernization creates new interfaces between subsystems which can then be replace one by one.

Applicability of the approach

The article is a good overview of modernization techniques and lists their strengths, weaknesses, targets, and use.

Constraints of the approach

Presented techniques are suitable for specific types of software and system environments.

Validation of the results
There is no validation, since the article is a survey of general black-box modernization techniques.

6.3.5 Warren & Ransom (2002)

In this article an overview of a method, and a process framework called Renaissance, is presented. It supports system evolution by first recovering a stable basis using reengineering, and subsequently continuously improving the system by a stream of incremental changes. The extent of evolution is determined by taking into account technical, business, and organizational factors.

Providing a controlled approach to system change means reducing the costs and risks. Key requirements of a method to support controlled system evolution are
- The method should support incremental evolution.
- Where appropriate the method should emphasize reengineering rather than system replacement.
- The method should prevent the legacy phenomena from reoccurring.
- It should be possible to customize the method to particular organizations and projects.

Process framework consists of two main phases:
- Evolution planning: "What to do", see detailed description in Ransom *et al.* (1998).
- Evolution project management: "How to do", contains implementation of evolution strategy and delivery and deployment of the products.

Applicability of the approach
The represented framework is highly relevant to some of the goals of ELTIS.

Constraints of the approach
Not explicated

Validation of the results
No actual validation. Authors have recieved a great deal of feedback and refinement suggestions from industrial partners of the RENAISSANCE project.

The key findings of companies which have evaluated RENAISSANCE were:
- Framework is well-defined and easy to follow.
- It integrates successfully with different project management processes.
- It helps risk reduction and cost distribution.
- Evolution strategy selection was proven useful.
- Overhead of adopting framework is high.
- Overhead for managing small projects was high.

Other related works
Other works on reengineering include Bailes & Peake (2003).

6.3.6 Harsu (2003)

Harsu's text book covers software maintenance and especially reengineering. Harsu (2000) has also written a Ph.D. thesis on the subject area. Following aspects are discussed in the book:

- Reverse engineering/design recovery.
- Software renovation/modernization (models of renovation, language conversion, wrapping).
- Data reengineering.
- Reusability improvements.

Applicability of the approach
The book describes approaches, which are highly relevant for the modernization support part of the project's objectives.

Validation of the results
Validation is not explicated, since the book describes existing techniques at non-detailed level.

6.3.7 Migration

The term migration refers here to the process of making legacy systems to function in new technical environment (Brodie & Stonebraker, 1995). Proactive actions towards system portability affect to the possibilities of cost-effective migration. Other recent works include Goedicke & Zdun (2002).

6.3.8 Restructuring

Restructuring (Griswold & Notkin, 1993) means the change of the internal representation of a system without changing the abstraction level of the representations (the external behaviour is neither affected).

6.3.9 Refactoring

Refactoring (Fowler *et al.*, 1999) means the general "polishing" of a system, often in conjunction to its changes for other reasons. The targets of development may be, *e.g.*: style, comprehensibility, flexibility or reusability.

6.3.10 Kataoka et al. (2002)

A (tool-supported) method, which helps finding out how *refactoring* will affect *maintainability* and choose correct refactorings to be carried out is presented. The method uses *coupling metrics* as a measure of maintainability.

Refactoring process consist of improvement planning, execution and validation. Planning begins with detecting "bad smell" (potential refactoring candidates: duplicate code, *etc*.) from source code. Then the candidates are analyzed and the most potential ones are selected to be included in a comprehensive refactoring plan. The plan produced is then evaluated in terms of cost and effect. Improvement execution starts with refactoring deployment as actual program modifications, which are then carried out and evaluated.

It's (still) totally unclear how this method helps to choose appropriate refactorings. It just magically happens during refactoring planning: "After analyzing various "bad-smells", a number of refactoring candidates would be identified." If it's (as one would assume) done by comparing metrics before and after refactoring, then where did the *after refactoring* -coupling metrics value come from *before* the refactoring was actually carried out. The paper doesn't talk about *estimated* metrics, only the actual ones.

Applicability of the approach
The only result applicable is the information: "there exists refactorings on which maintainability measured by expert judgment and coupling metrics correlate." The method presented doesn't seem applicable at all for various reasons. It's totally, completely and entirely unclear how the refactoring candidates are selected from "bad smells" and how cost-effect-evaluation of improvement plan is actually carried out. The number of refactorings whose effect can be evaluated from coupling metrics viewpoint is limited. In "validation" the refactorings selected could well have been the ones whose effect is measurable by coupling metrics.

Constraints of the approach
Tool support (*Refactoring Assistant*) is required.

Validation of the results
There is no actual validation. A single experiment with single 5 year old software project (size unknown) written in C++ was used as a case study.

Related works
Visaggio (2000) may be relevant.

6.3.11 Redocumentation

Redocumentation means the creation of a new more illustrative representation for the system, which still is equivalent to the original one. In effect, the aim is to create the documentation,

which should have been created in the past (look *e.g.*: Antoniol *et al.*, 2000). Look also Prechelt *et al.* (2002).

6.4 Reverse engineering

Reverse engineering means the identification and representation of system components and their interrelations, in a new form, which typically is at a higher abstraction level than the original one (Cross *et al.*, 1992; Lano & Haughton, 1993). Thus reverse engineering produces representation transformations (Bennett, 1998), which support *e.g.* comprehensibility (von Mayrhauser & Vans, 1995). Reverse engineering usually aims at *design recovery* (look *e.g.* Gannod & Cheng, 1999; Niere *et al.*, 2002). A thorough and extensive classification of the reverse engineering techniques is found in (Koskinen, 2000; Introduction and overview, Appendix I).

The applicability of reverse engineering tools is limited by their availability to the needed platform, operating system and programming language. Old technologies (which often are most troublesome) are not necessarily well supported. Some representative examples of the modern, versatile tools of this category, which also support a relatively wide range of programming languages include: *Telelogic LogiScope*, *Imagix 4D*, *SNiFF+*, and *Refine/C* (look *e.g.* Bellay & Gall, 1997, for comparisons). Look also Chen *et al.* (1995).

6.4.1 Information request specifications

Information requests, which the maintainer formulates initiate the analysis operations of the support tool. The specification mechanisms affect the possibilities of sufficiently exact and detailed queries. There exists good mechanisms for these purposes, although they are still at research stage. *E.g.* Paul & Prakash (1996) have proposed a dedicated query language for this purpose.

6.4.2 Impact analysis support

The purpose of impact analysis (Queille *et al.*, 1994; Arnold & Bohner, 1996) is to identify the possible side-effects of code changes. In principle each code change should be followed by regression testing, which would guarantee that the changed system still meets all the requirements. In practice, in case of large systems, it is not possible to perform complete impact analysis related to all changes. Focusing of impact analysis may be supported (in principle) *e.g.* by program slicing, especially its static form.

6.4.2.1 Program slicing

One of the approaches which is theoretically very well suited especially for maintenance support, is program slicing (Weiser, 1982; Berzins, 1995; Kamkar, 1995). The most important of the program dependencies which should be checked while programs are changed are data- and control flow dependencies (Paakki *et al.*, 1997). Program slices are formed

based on these dependencies. The main restriction of program slicing is its limited efficiency in case of analyzing industry-size programs.

It is also important to identify (related to code changes) those parts of the code which have been changed or which are not changed ("frozen parts"), or on only which necessary preconditions changes should be allowed (invariants) (look *e.g.* Ernst, 2001).

6.4.3 Program visualization

The results of the reverse engineering are represented to the user either in text-form or graphically. Hierarchical and graphical representations typically aid in the comprehension of large information collections/structures (Ball & Eick, 1996; Eick *et al.*, 2002; Yin & Keller, 2002).

6.4.4 Reverse engineering of object-oriented software

As an application area of reverse engineering legacy systems are very important. However, some effort has also been targeted to the support of more modern object-oriented programs, which are more and more important in the future (Chen *et al.*, 1998; Systä, 2000; Systä *et al.*, 2001; Ferenc *et al.*, 2002).

6.5 Preventive actions for maintenance

6.5.1 Enhancement of maintainability during implementation phase

Often proactive, preventive actions towards higher maintainability (Smith, 1999) of systems is cost-effective. The possibilities for this are affected *e.g.* by the applied programming languages. *E.g.* for enhancing correctness (and thus reducing needed effort for future corrective maintenance) some languages provide such mechanisms as: *defensive programming* (error recovery), *pre/post conditions* of procedures, *assertions* (*e.g.* Binder (2001); Müller *et al.*, 2002) and *tracing mechanisms.*

6.5.2 Enhancement of maintainability during design phase

In principle, maintainability should be taken into account already in the design phase of the system by trying to anticipate the probable changes (look *e.g.* Koskimies, 1997). This is called *speculative design* and it should, in one form or another, be noted during whole software quality assurance process. Look also Baxter & Pidgeon (1997); Schach & Tomer (2000) on the subject.

7 CONCLUSIONS

This report has summarized the theoretical background studies of the ELTIS project. Software maintenance is generally an undervalued area of research and development, while taking into account its actual importance. It causes most of the total software costs (often 50-75% in case of successful systems with long lifetime). 1) Thus effective process improvements which take into account the effects on maintainability appear beneficial in long-term. 2) Maintenance appears to be (in general) a constant problem, despite the earlier process improvements and development of system design methodologies (most likely because of the constant increases of system sizes and program complexity). 3) These two points underline the importance of paying attention to this area and identifying the actual cost drivers, whose improvement could have substantial effect on the process profitability.

Software maintenance is often also quite demanding (despite the common view of it being more or less routine work), especially in case of maintaining large legacy systems. These systems are typically hard to maintain, but they cannot be replaced because of their great business value and the application domain knowledge that they contain. Thus they tend to have long lifetime. Lehman (1998) has gathered longitudinal empirical evidence on large-scale industry-level software development and derived general laws of system evolution described in this report.

Software maintenance is further complicated in case of large tasks of adaptive maintenance category (*i.e.* software modifications made due to changing technical environment) without proper system documentation (or other useful information sources) available. In an ideal situation the existing code could be reused and modified flexibly by the original developers. That would support the process of gaining the original investment back, and thus to improve return on investment (potentially for both the organization developing the software and the one using it). It is clear that this goal is easiest to achieve via long-term maintenance process improvement and preventive maintenance.

However, in case of answering to the acute problems of dealing with current legacy systems, other support processes are required. One of the central questions is whether to continue maintaining the existing legacy systems or not. Their modernization might be an option. Another (typically less favorable) main possibility might be their complete rewrite. Modernization may require integration of the old (legacy) system to newer applications (Robertson, 1997; Coyle, 2000) or its encapsulation (Sneed, 2000).

There exists a lot of general literature related to the (relatively scattered) preliminary objectives set to the project. Estimating costs of building a new software system is an extensively covered subject area. The main cost driver for this is the required effort (measured in man-months). However, even the best of the current software cost models (*e.g.* COCOMO) have relatively large error-span, even in case that the models would be backed with a relatively large project database containing actual empirical data and properly calibrated to take into account the

factors characterizing the system development process of the organization developing the system.

In the area of software maintenance cost estimation, the situation is even worse in this regard. In an (theoretical) ideal case there would exist a computer program (a decision support system) which would receive as input parameters the central decision criteria for software replacement or modernization, provide the answer to the question, and outline the arguments. Although there exists many theoretical works on the software maintenance cost/effort estimation area, there are no generally accepted or rigorously and successfully empirically validated models. Most of the existing models either make unrealistic simplifications or assume existence of versatile metrics data (or both).

One of the general problems related to this part of software engineering is the lack of really reliable metrics for software complexity (which, however, is a prime candidate as a prime cost driver). Main reference to this area is Kemerer (1995). The often used LOC, FP and cyclomatic complexity measures all have severe limitations and in some cases their use may provide completely misleading results. General collection of (technical and process) metrics data is discussed *e.g.* by Grady (1994). In reality there exists a great many factors which should be taken into account while deciding about system modernization. Many of these factors are not even related to the metrics data which is typically collected or collectable.

Because of the reasons outlined above, there is a great need (both because of theoretical and practical reasons), to gather versatile empirical data on the actual system portfolios, system characteristics and actual expert decision making processes related to software modernization. Kemerer & Slaughter (1999) have outlined the following criteria for successful empirical software maintenance research: 1) a large data source from programs and versions, 2) willingness of a good commercial partner to participate in the research project, and 3) highly disciplined research approach with desire to expand the previous research.

Sahin's & Zahedi's (2001) work is also important since it is empirically well validated and pays attention on customer satisfaction related to software upgrade cycle. It provides a model for strategic decision making in software maintenance, dealing with warranties, maintenance, and upgrades. The model can be used in analyzing organizations' system portfolios.

Gode *et al.* (1990) and Chan *et al.* (1996) are the only ones really trying to say something about actual software replacement timing. These models are mostly formal exercises, with very limited real validation. The problems that the developers of these models have faced could be interpreted as indication of inherent complexity of the replacement decision. There is also the problem of acquiring reliable required input data for these models.

Sneed's works (1995a; 1995b) are also interesting (despite the problems with their proprietary nature, assumed availability of metrics data, and limited empirical validation). He has given both a model for general software maintenance cost estimation (1995a) (discussing maintainability factors, and maintenance scope determination) and reengineering cost

estimation (1995b) (taking into account software business considerations, such as return on investment and enhanced business value, and providing a long list of potentially useful measurements).

It appears that all closed-form and black-box solutions providing direct decisions (assumed as being optimal) are bound to fail due to the versatility of non-measurable factors and unpredictable side-conditions. Providing supporting information to the expert who actually makes the decision seems to be the right way to continue.

Also the main branches of technical support solutions for software modernization are outlined and references to main works given. Harsu's (2003) text-book, Warren & Ransom (2002), Koskinen (2000), Fowler *et al.* (1999), and Arnold (1993) are some of the main references in the subareas of technical modernization support. The above-cited Sneed's (1995b) work also relates directly to this area.

8 REFERENCES

1) Abran, A. & Robillard, P. (1993). "Reliability of function points productivity model for enhancement projects (a field study)". *Conference on Software Maintenance 1993*, 80-97. IEEE Computer Society Press.

2) Abran, A., Silva, I. & Primera, L. (2002). "Field studies using functional size measurement in building estimation models for software maintenance". *Journal of Software Maintenance and Evolution: Research and Practice* **14**, 31-64.

3) Agans, D. (2002). "*Debugging: The Nine Indispensable Rules for Finding Even the Most Elusive Software and Hardware Problems*". AMACOM, 192 p.

4) Albrecht, A. & Gaffney, J. (1983). "Software function, source lines of code, and development effort prediction: a software science validation". *IEEE Transactions on Software Engineering* **SE-9** (6), 639-648.

5) Antoniol, G., Canfora, G., Casazza, G. & De Lucia, A. (2000). "Information retrieval models for recovering traceability links between code and documentation". *Proceedings of the International Conference on Software Maintenance - 2000*, 40-49. IEEE Computer Soc.

6) Arnold, R. (1993). "*Software Reengineering (IEEE Computer Society Press Tutorial)*". IEEE Computer Society, 675 p.

7) Arnold, R. & Bohner, S. (Eds.) (1996). "*Software Change Impact Analysis*". Wiley-IEEE Press, 392 p.

8) Bailes, P. & Peake, I. (2003). "Incremental enhancement of the expressiveness of a reengineering tool development platform". *Proceedings of the 21st IASTED International Conference, APPLIED INFORMATICS*, 927-934.

9) Ball, T. & Eick, S. (1996). "Software visualization in the large". *Computer* **29** (4), 33-43.

10) Baniassad, E. & Murphy, G. (1998). "Conceptual module querying for software reengineering". *Proceedings of the 1998 International Conference on Software Engineering (ICSE'98)*, 64-73. IEEE Computer Soc.

11) Banker, R., Datar, S. & Kemerer, C. (1991). "A model to evaluate variables impacting the productivity of software maintenance projects". *Management Science* **37** (1).

12) Banker, R., Datar, S., Kemerer, C. & Zweig, D. (1993). "Software complexity and maintenance costs". *Communications of the ACM* **36** (11), 81-94.

13) Banker, R. & Slaughter, S. (1994). "Project size and software maintenance productivity: empirical evidence on economics of scale in software maintenance". In: DeGross, J., Huff, S. & Munro, M. (Eds.) *Proceedings of the Fifteenth International Conference on Information Systems*, 279-289. Distributed by ACM.

14) Basili, V. (1990). "Viewing maintenance as reuse-oriented software development". *IEEE Software* **7** (1), 19-25.

15) Baxter, I. & Pidgeon, C. (1997). "Software change through design maintenance". *Proceedings of the International Conference on Software Maintenance - 1997*, 250-259. IEEE Computer Soc.

16) Bellay, B. & Gall, H. (1997). "A comparison of four reverse engineering tools". *Proceedings of the 4th Working Conference on Reverse Engineering*, 2-11. Los Alamitos, CA: IEEE Computer Society.

17) Bennett, K. (1998). "Do program transformations help reverse engineering?". *Proceedings of the International Conference on Software Maintenance - 1998*, 247-254. IEEE Computer Soc.

18) Bennett, M. & Gittens, M. (1997). "Empirical defect modeling to extend the Constructive Cost Model". *The Eight European Software Control and Metrics Conference (ESCOM'97)*. Conf. location: Berlin, Germany.

19) Bennett, K., Lientz, B. & Swanson, E. (1980). "*Software Maintenance Management*", Addison Wesley.

20) Berlack, R. (1991). "*Software Configuration Management*" (Wiley Series in Software Engineering Practice). John Wiley & Sons, 352 p.

21) Berzins, V. (Ed.) (1995). "*Software Merging and Slicing*". IEEE Computer Soc.

22) Bianchi, A., Caivano, D., Marengo, V. & Visaggio, G. (2003). "Iterative reengineering of legacy systems". *IEEE Transactions on Software Engineering* **29** (3), 225-241.

23) Binder, R. (2001). "*Testing Object-Oriented Systems: Models, Patterns, and Tools*" (3rd printing). Addison-Wesley.

24) Bisbal, J., Lawless, D., Wu, B. & Grimson, J. (1999). "Legacy information systems: issues and directions". *IEEE Software* **16** (5), 103-111.

25) Bitman, W. (1999). "A metrics-based decision support tool for software module interfacing technique selection to lower maintenance cost". *Sixth IEEE International Symposium on Software Metrics*, 170-178.

26) Boehm, B. (1981). "*Software Engineering Economics*"**,** Prentice Hall, 1981.

27) Boehm, B. (1984). "Software engineering economics". *IEEE Transactions on Software Engineering* **10** (1), 4-21.

28) Boehm, B., Horowitz, E., Madachy, R., Reifer, D., Clark, B., Steece, B., Brown, A.W., Chulani, S. & Abts, C. (2000). "*Software Cost Estimation with COCOMO II*". Prentice Hall, 502 p.

29) Boehm, B. & Papaccio, P. (1988). "Understanding and controlling software costs". *IEEE Transactions on Software Engineering* **14** (10), 1462-1477.

30) Bray, O. & Hess, M. (1995). "Reengineering a configuration management system". *IEEE Software* **12** (1), 55-63.

31) Bredero, R., Hupkes, E. & Pagrach, D. (1995). "Modelling maintenance cost from factual data: a practical example". *European Software Cost Modelling Conference (ESCOM'95)*. Conf. location: Rolduc Abbey, The Netherlands.

32) Briand, L., Basili, V., & Thomas (1992). "A pattern recognition approach for software engineering analysis". *IEEE Transactions on Software Engineering* **18** (11), 931-942.

33) Briand, L., El Emam, K., Surmann, D., Wieczorek, I. & Maxwell, K. (1999). "An assessment and comparison of common software cost estimation modeling techniques". *Proceedings of the 1999 International Conference on Software Engineering*, 313-322.

34) Briand, L., Langley, T. & Wieczorek, I. (2000). "A replicated assessment and comparison of common software cost modeling techniques". *Proceedings of the 2000 International Conference on Software Engineering (ICSE 2000)*, 377-386. ACM Press.

35) Brodie, M.L. & Stonebraker, M. (1995). "*Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach*". Morgan Kaufmann, 210 p.

36) Burd, E. & Munro, M. (1997). "Investigating the maintenance implications of the replication of code". *Proceedings of the International Conference on Software Maintenance – 1997*, 322-330. IEEE Computer Soc.

37) Caivano, D., Lanubile, F. & Visaggio, G. (2001). "Software renewal process comprehension using dynamic effort estimation". *Proceedings of the IEEE International Conference on Software Maintenance - 2001*, 209-218. IEEE Computer Soc.

38) Calzolari, F., Tonella, P. & Antoniol, G. (1998). "Dynamic model for maintenance and testing effort". In: Khoshgoftaar, T. & Bennett, K. (Eds.) *Proceedings of the International Conference on Software Maintenance - 1998*, 104-112. IEEE Computer Soc.

39) Capretz, M. & Munro, M. (1994). "Software configuration management issues in the maintenance of existing systems". *Software Maintenance: Research & Practice* **6** (1), 1-14.

40) Chapin, N., Hale, J., Khan, K., Ramil, J. & Tan, W.-G. (2001). "Types of software evolution and software maintenance". *Journal of Software Maintenance and Evolution: Research & Practice* **13** (1), 3-30.

41) Chan, T., Chung, S. & Ho, T. (1994). "Timing of software replacement". In: DeGross, J., Huff, S. & Munro, M. (Eds.) *Proceedings of the Fifteenth International Conference on Information Systems*, 291-307.

42) Chan, T., Chung, S. & Ho, T. (1996). "An economic model to estimate software rewriting and replacement times". *IEEE Transactions on Software Engineering* **22** (8), 580-598.

43) Chen, Y.-F., Fowler, G., Koutsofios, E. & Wallach, R. (1995). "Ciao: a graphical navigator for software and document repositories", *Proceedings of the International Conference on Software Maintenance - 1995*, 66-75. IEEE Computer Soc.

44) Chen, Y.-F., Gansner, E. & Koutsofios, E. (1998). "A C++ data model supporting reachability analysis and dead code detection". *IEEE Transactions on Software Engineering* **24** (9), 682-694.

45) Chu, W., Lu, C.-W, Shiu, C.-P. & He, X. (2000). "Pattern-based software reengineering: a case study". *Journal of Software Maintenance and Evolution: Research and Practice* **12**, 121-141.

46) Coleman, D., Ash, D., Lowther, B. & Oman, P. (1994). "Using metrics to evaluate software system maintainability". *Computer* **27** (8), 44-49.

47) Comella-Dorda, S., Wallnau, K., Seacord, R. & Robert, J. (2000). "A survey of black-box modernization approaches for information systems". *Proceedings of the International Conference on Software Maintenance - 2000*, 173-183. IEEE Computer Soc.

48) Coyle, F. (2000). "Legacy integration changing perspectives". *IEEE Software* **17** (2), 37-41.

49) Cross, J.H. II, Chikofsky, E. & May, C.H. Jr. (1992). "Reverse engineering". Yovits, M. (Ed.) *Advances in Computers* **35**, 199-254. Academic Press.

50) De Lucia, A., Di Penta, M., Stefanucci, S. & Venturi, G. (2002). "Early effort estimation of massive maintenance processes". *Proceedings of the International Conference on Software Maintenance - 2002*, 234-237. IEEE Computer Soc.

51) De Lucia, A., Pannella, A., Pompella, E. & Stefanucci, S. (2001). "Assessing massive maintenance processes: an empirical study". *Proceedings of the IEEE International Conference on Software Maintenance*, 451-458. IEEE Computer Soc.

52) Di Lucca, G., Di Penta, M. & Gradara, S. (2002). "An approach to classify software maintenance requests". *Proceedings of the International Conference on Software Maintenance - 2002*, 93-102. IEEE Computer Soc.

53) Edelstein, D. (1993). "Report on the IEEE STD 1219-1993 – Standard for Software Maintenance". *ACM SIGSOFT Software Engineering Notes* **18** (4), p. 94.

54) Eick, S., Graves, T., Karr, A., Mockus, A. & Schuster, P. (2002). "Visualizing software changes". *IEEE Transactions on Software Engineering* **28** (4), 396-412.

55) Eisenstadt, M. (1997). "My hairiest bug war stories". *Communications of the ACM* **40** (4), 30-37.

56) Engelhart, J. (1995). "FPA and maintenance". *European Software Cost Modelling Conference (ESCOM'95)*. Conf. location: Rolduc Abbey, The Netherlands.

57) Ernst, M., Cockrell, J., Griswold, W. & Notkin, D. (2001). "Dynamically discovering likely program invariants to support program evolution". *IEEE Transactions on Software Engineering* **27** (2), 99-123.

58) Fanta, R. & Rajlich, V. (1998). "Reengineering object-oriented code". *Proceedings of the International Conference on Software Maintenance - 1998*, 238-246. IEEE Computer Soc.

59) Fanta, R. & Rajlich, V. (1999). "Removing clones from the code". *Journal of Software Maintenance: Research and Practice* **11**, 223-243.

60) Fasolino, A., Natale, D., Poli, A. & Quaranta, A. (2000). "Metrics in the development and maintenance of software: an application in a large scale environment". *Journal of Software Maintenance: Research and Practice* **12**, 343-355.

61) Feiler, J. & Butler, B. (1999). "*Y2K Bible*". Hungry Minds, 541 p.

62) Fenton, N. (1994). "Software measurement: a necessary scientific basis". *IEEE Transactions on Software Engineering* **20** (3), 199-206.

63) Ferenc, R., Beszedes, A., Tarkiainen, M. & Gyimothy, T. (2002). "Columbus - reverse engineering tool and schema for C++". *Proceedings of the International Conference on Software Maintenance - 2002*, 172-181. IEEE Computer Soc.

64) Foster (1991). "Program lifetime: a vital statistic for maintenance". *IEEE Proceedings of the International Conference on Software Maintenance*, 98-103. IEEE Computer Soc.

65) Foster & Kiekuth (1990). "Software maintenance survey: summary". BT Laboratories.

66) Fowler, M., Beck, K., Brant, J., Opdyke, W. & Roberts, D. (1999). *"Refactoring: Improving the Design of Existing Code*". Addison-Wesley, 431 p.

67) Furey, S. (1997). "Why we should use function points". *IEEE Software* **14** (2), 28-31.

68) Gannod, G. & Cheng, B. (1999). "A framework for classifying and comparing software reverse engineering and design recovery tehcniques". *Proceedings of the Sixth Working Conference on Reverse Engineering*, 77-88. IEEE Computer Soc.

69) Gerlich, R. & Denskat, U. (1994), "A cost estimation model for maintenance and high reuse". *Proceedings of the European Software Cost Modelling Meeting (ESCOM'94)*. May 11-13, 1994. Conf. location: Ivrea, Italy.

70) Gibson, V. & Senn, J. (1989). "System structure and software maintenance performance". *Communications of the ACM* **32** (3), 347-358.

71) Gill, G. & Kemerer, C. (1991). "Cyclomatic complexity density and software maintenance productivity". *IEEE Transactions on Software Engineering* **17** (12), 1284-1288.

72) Gode, D., Barua, A. & Mukhopadhyay, T. (1990). "On the economics of the software replacement problem". In: DeGross, J., Alavi, M. & Oppelland, H. (Eds.) *Proceedings of the Eleventh International Conference on Information Systems*, 159-170.

73) Goedicke, M. & Zdun, U. (2002). "Piecemeal legacy migrating with an architectural pattern language: a case study". *Journal of Software Maintenance and Evolution: Research and Practice* **14**, 1-30.

74) Gorla, N., Benander, A. & Benander, B. (1990). "Debugging effort estimation using software metrics". *IEEE Transactions on Software Engineering* **16** (2), 223-231.

75) Grady, R. (1994). "Successfully applying software metrics". *Computer* **27** (9), 18-25.

76) Griswold, W. & Notkin, D. (1993). "Automated assistance for program restructuring". *ACM Transactions on Software Engineering and Methodology* **2** (3), 228-269.

77) Harsu, M. (2000). "*Re-engineering Legacy Software Through Language Conversion*" (Ph.D. thesis). Department of Computer and Information Sciences, University of Tampere.

78) Harsu, M. (2003). "*Ohjelmien ylläpito ja uudistaminen*" (in Finnish). Talentum, 292 p.

79) Haug, M., Olsen, E. & Cuevas, G. (Eds.) (2001). "*Managing the Change: Software Configuration & Change Management*". Springer Verlag, 297 p.

80) Henry, J., Blasewitz, R. & Kettinger, D. (1996). "Defining and implementing a measurement-based software maintenance process". *Software Maintenance: Research and Practice* **8**, 79-100.

81) Hürten R. *et al*. (1996). "Estimating the effort of maintenance and enhancement". *The Seventh European Software Control and Metrics Conference (ESCOM'96)*. Conf. location: Wilmslow, Cheshire, UK.

82) ITV (2000). "Y2K maksoi Nokialle 450 miljoonaa" (in Finnish). *IT Viikko* (10.2.2000).

83) Jambor-Sadeghi, K., Ketabchi, M., Chue, J. & Ghiassi, M. (1994). "A systematic approach to corrective maintenance". *The Computer Journal* **37** (9), 764-778.

84) Jones, C. (1997). "Slow response to Year 2000 problem". *IEEE Software* **14** (3), 114-115 (an interview).

85) Jørgensen, M. (1995). "An empirical study of software maintenance tasks". *Software Maintenance: Research and Practice* **7**, 27-48.

86) Jørgensen, M. & Sjøberg, D. (2002). "Impact of experience on maintenance skills". *Journal of Software Maintenance: Research and Practice* **14**, 123-146.

87) Kafura, D. & Reddy, G. (1987). "The use of software complexity metrics in software maintenance". *IEEE Transactions on Software Engineering* **SE-13** (3), 335-343.

88) Kajko-Matsson, M. (2002). "Problem management maturity within corrective maintenance". *Journal of Software Maintenance and Evolution: Research and Practice* **14**, 197-227.

89) Kamkar, M. (1995). "An overview and comparative classification of program slicing techniques". *The Journal of Systems and Software* **31** (3), 197-214.

90) Kataoka, Y., Imai, T., Andou, H. & Fukaya, T. (2002). "A quantitative evaluation of maintainability enhancement by refactoring". *Proceedings of the International Conference on Software Maintenance - 2002*, 576-585. IEEE Computer Soc.

91) Kemerer, C. (1987). "An empirical validation of software cost estimation models". *Communications of the ACM* **30** (5), 416-429.

92) Kemerer, C. (1995). "Software complexity and software maintenance: a survey of empirical research". *Annals of Software Engineering* **1**, 1-22. J.C. Baltzer AG, Science Publishers.

93) Kemerer, C. & Slaughter, S. (1999). "An empirical approach to studying software evolution". *IEEE Transactions on Software Engineering* **25** (4), 493-509.

94) Kiran, G., Haripriya, S. & Jalote, P. (1997). "Effect of object orientation on maintainability of software", *Proceedings of the International Conference on Software Maintenance - 1997*, 114-112. IEEE Computer Soc.

95) Kitchenham, B. (1997). "The problems with function points". *IEEE Software* **14** (2), 28-31.

96) Kitchenham, B., Pfleeger, S., Pickard, L., Jones, P., Hoaglin, D., El Emam, K. & Rosenberg, J. (2002). "Preliminary guidelines for empirical research in software engineering". *IEEE Transactions on Software Engineering* **28** (8), 721-734.

97) Kitchenham, B. & Taylor, N. (1984). "Software cost models". *ICL Technical Journal* **4** (1), 73-102.

98) Koskimies, K. (1997). "Pieni oliokirja" (in Finnish). Suomen Atk-kustannus.

99) Koskinen, J. (2000). "*Automated Transient Hypertext Support for Software Maintenance*". *Jyväskylä Studies in Computing* **4**. University of Jyväskylä.

100) Laitinen, K. (1995). "*Natural naming in software development and maintenance*" (Ph.D. thesis). VTT.

101) Landsbaum, J.B., Glass, R.L. & Glass, R.B. (1992). "*Measuring and Motivating Maintenance Programmers*". Prentice Hall, 96 p.

102) Lano, K. & Haughton, H. (1993). "*Reverse Engineering and Software Maintenance: A Practical Approach*" (McGraw-Hill International Series in Software Engineering). McGraw-Hill.

103) Lanning, D. & Khoshgoftaar, T. (1994). "Modeling the relationship between source code complexity and maintenance difficulty". *Computer* **27** (9), 35-40.

104) Lehman, M.M. & Belady, L.A. (1985). "*Program Evolution: Processes of Software Change*" (Apic Studies in Data Processing). Academic Press.

105) Lehman, M., Perry, D. & Ramil, J. (1998). "Implications of evolution metrics on software maintenance". *Proceedings of the International Conference on Software Maintenance - 1998*, 208-217. IEEE Computer Soc.

106) Leon, A. (2000). "*A Guide to Software Configuration Management*" (Artech House Computer Library). Artech House, 384 p.

107) Letovsky, S. & Soloway, E. (1986). "Delocalized plans and program comprehension". *IEEE Software* **3** (3), 41-49.

108) Lientz, B.P. & Swanson, E. (1980). "*Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*". Addison-Wesley: Reading, MA, 214 p.

109) Lyon, D. (1999). "*Practical CM: Best Practices for the 21st Century*" (2nd ed.). Raven, 260 p.

110) Mancini, L. & Ciampoli, R. (1990). "Maintenance cost estimation: industrial point of view". *1990 European COCOMO User's Group Meeting*. Conf. location: Botley, Hampshire, UK.

111) Martin, J. (1983). "*Software Maintenance: The Problem and Its Solution*". Prentice Hall, 472 p.

112) von Mayrhauser, A. (1994). "Maintenance and evolution of software products". *Advances in Computers* **39**, 1-49.

113) von Mayrhauser, A. & Vans, A. (1995). "Program understanding models and experiments". *Advances in Computers* **40**, 1-38.

114) McCabe, T. (1976). "A complexity measure". *IEEE Transactions on Software Engineering* **2** (4), 308-320.

115) Miller, H. (1998). "*Reengineering Legacy Software Systems*". Digital Press, 250 p.

116) Moreton (1988). "Analysis and results from a maintenance survey". *Proceedings of the 2nd Software Maintenance Workshop*.

117) Munson, J. & Elbaum, S. (1998). "Code Churn: a measure for estimating the impact of code change". In: Khoshgoftaar, T. & Bennett, K. (Eds.) *Proceedings of the International Conference on Software Maintenance - 1998*, 24-31. IEEE Computer Soc.

118) Müller, M., Typke, R. & Hagner, O. (2002). "Two controlled experiments concerning the usefulness of assertions as a means for programming". *Proceedings of the International Conference on Software Maintenance – 2002*, 84-92. IEEE Computer Soc.

119) Niere, J., Schäfer, W., Wadsack, J., Wendehals, L. & Welsh, J. (2002). "Towards pattern-based design recovery". *International Conference on Software Engineering (ICSE'02)*, 338-348.

120) Niessink, F. & van Vliet, H. (1997). "Predicting maintenance effort with function points". *Proceedings of the International Conference on Software Maintenance - 1997*, 32-39. IEEE Computer Soc.

121) Niessink, F. & van Vliet, H. (1998). "Two case studies in measuring software maintenance effort". *Proceedings of the International Conference on Software Maintenance - 1998*, 76-85. IEEE Computer Soc.

122) Nosek & Palvia (1990). "Software maintenance management: changes in the last decade". *Journal of Software Maintenance: Research and Practice* **2** (3), 157-174.

123) Oman, P. & Cook, C. (1990). "Typographic style is more than cosmetic". *Communications of the ACM* **33** (5), 506-520.

124) Oman, P. & Cook, C. (1991). "A programming style taxonomy". *The Journal of Systems and Software* **15** (3), 287-301.

125) Oman, P. & Hagemeister, J. (1992). "Metrics for assessing a software system's maintainability". *Proceedings of the 1992 Software Maintenance Conference*, 337-344.

126) Oman, P. & Hagemeister, J. (1994). "Construction and testing of polynomials predicting software maintainability". *Journal of Systems and Software* **24** (3), 251-266.

127) Paakki, J., Koskinen, J. & Salminen, A. (1997). "From relational program dependencies to hypertextual access structures". *Nordic Journal of Computing* **4** (1), 3-36.

128) Paul, S. & Prakash, A. (1996). "A query algebra for program databases". *IEEE Transactions on Software Engineering* **22** (3), 202-217.

129) Pearse, T. & Oman, P. (1995). "Maintainability measurements on industrial source code maintenance activities". *Proceedings of the International Conference on Software Maintenance - 1995*, 295-303. IEEE Computer Soc.

130) Phua, P.K.H. (2002). "*Software engineering economics*", Chapters 8 (Software cost-estimation methods & procedures), 9 (Software maintenance and life-cycle cost estimation) (lecture notes). Department of Information Systems, School of Computing, National University of Singapore, Singapore.

131) Pigoski, T.M. (1996). "*Practical Software Maintenance: Best Practices for Managing Your Software Investment*". John Wiley & Sons, 384 p.

132) Polo, M., Piattini, M. & Ruiz, F. (2001). "Using code metrics to predict maintenance of legacy programs: a case study". *Proceedings of the IEEE International Conference on Software Maintenance - 2001*, 202-208. IEEE Computer Soc.

133) Polo, M., Piattini, M., Ruiz, F. & Mohammadian, M. (Eds.) (2003). "*Advances in Software Maintenance Management: Technologies and Solutions*" (to be published?).

134) Prechelt, L., Unger-Lamprecht, B., Philippsen, M. & Tichy, W. (2002). "Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance". *IEEE Transactions on Software Engineering* **28** (6), 595-606.

135) Pressman, R. (2001). "Software process and project metrics" (Chapter 4). In: *Software Engineering - A Practitioner's Approach (5th ed.)*, 79-111.

136) Pressman, R. (2001). "Technical metrics for software" (Chapter 19). In: *Software Engineering - A Practitioner's Approach (5th ed.)*, 507-538.

137) Pressman, R. (2001). "Technical metrics for object-oriented systems" (Chapter 24). In: *Software Engineering - A Practitioner's Approach (5th ed.)*, 653-669.

138) Queille, J.P., Voidrot, J.-F., Wilde, N. & Munro, M. (1994). "The impact analysis task in software maintenance: a model and a case study". *Proceedings of the International Conference on Software Maintenance* (*ICSM'94*).

139) Ramil, J. & Lehman, M. (2000). "Metrics of software evolution as effort predictors: a case study". *Proceedings of the International Conference on Software Maintenance*, 163-172. IEEE Computer Soc.

140) Riecken, R., Koenemann-Belliveau, J. & Robertson, S. (1991). "What do expert programmers communicate by means of descriptive commenting". J. Koenemann-Belliveau, T. Moher & S. Robertson (Eds.): *Empirical Studies of Programmers: 4th Workshop* (*ESP'91*), 177-195. Norwood, NJ: Ablex.

141) Robertson, P. (1997). "Integrating legacy systems with modern corporate applications". *Communications of the ACM* **40** (5), 39-46.

142) Rombach, H. (1991). "Software reuse: a key to the maintenance problem". *Information and Software Technology* **33** (1), 86-92.

143) Rose, E. & Eriksson, I. (1998). "Development and maintenance costs: measures of software maintainability". In: Carlsson & Eriksson (Eds.) *Global & Multiple Criteria Optimization and Information Systems Quality*, 21-35. Åbo Akademi tryckeri.

144) Sahin, I. & Zahedi, M. (2001). "Policy analysis for warranty, maintenance, and upgrade of software systems". *Journal of Software Maintenance and Evolution: Research and Practice* **13**, 469-493.

145) Schach, S. & Tomer, A. (2000). "A maintenance-oriented approach to software construction". *Journal of Software Maintenance: Research and Practice* **12**, 25-45.

146) Schneidewind, N. (1997). "Measuring and evaluating maintenance process using reliability, risk, and test metrics". *Proceedings of the International Conference on Software Maintenance - 1997*, 232-239. IEEE Computer Soc.

147) Seaman, C. (2002). "The information gathering strategies of software maintainers". *Proceedings of the International Conference on Software Maintenance - 2002*, 141-149. IEEE Computer Soc.

148) Sheldon, F., Jerath, K. & Chung, H. (2002). "Metrics for maintainability of class inheritance hierarchies". *Journal of Software Maintenance and Evolution: Research and Practice* **14**, 147-160.

149) Singer, J. (1998). "Practices of software maintenance". *Proceedings of the International Conference on Software Maintenance - 1998*, 139-145. IEEE Computer Soc.

150) Smith, D.D. (1999). "*Designing Maintainable Software*". Springer Verlag, 169 p.

151) Sneed, H. (1995a). "Estimating the costs of software maintenance tasks". *Proceedings of the International Conference on Software Maintenance - 1995*, 168-181. IEEE Computer Soc. Press.

152) Sneed, H. (1995b). "Planning the reengineering of legacy systems". *IEEE Software* **12** (1), 24-34.

153) Sneed, H. (1999). "Risks involved in reengineering projects". *Proceedings of the IEEE Sixth Working Conference on Reverse Engineering*, 204-211.

154) Sneed, H. (2000). "Encapsulation of legacy software: a technique for reusing legacy software components". *Annals of Software Engineering* **9**, 293-313.

155) Sommerville, I. (1996). *Software Engineering (5th ed.)*. Addison-Wesley.

156) Sommerville, I. (1996). "Software cost estimation" (Chapter 29). In: *Software Engineering (5th ed.)*, 589-610. Addison-Wesley.

157) Stark, G., Oman, P., Skillicorn, A. & Ameele, A. (1999). "An examination of the effects of requirements changes on software maintenance releases". *Journal of Software Maintenance: Research and Practice* **11**, 293-309.

158) Swanson, E.B. & Beath, C.M. (Eds.) (1989). "*Maintaining Information Systems in Organizations*". John Wiley & Sons.

159) Swanson, E. & Beath, C. (1990). "Departmentalization in software development and maintenance". *Communications of the ACM* **33** (6), 658-667.

160) Systä, T. (2000). "*Static and Dynamic Reverse Engineering Techniques for Java Software Systems*" (Ph.D. thesis). Department of Computer Science and Information Sciences, University of Tampere, 233 p.

161) Systä, T., Koskimies, K. & Müller, H. (2001). "Shimba - an environment for reverse engineering Java software systems". *Software - Practice and Experience* **31**, 371-394.

162) Tahvildari, L. & Kontogiannis, K. (2002). "A software transformation framework for quality-driven object-oriented re-engineering". *Proceedings of the International Conference on Software Maintenance - 2002*, 596-605. IEEE Computer Soc.

163) Takang, A.A. & Grubb, P.A. (1996). "*Software Maintenance: Concepts and Practice*". International Thomson.

164) Tamai, T. & Torimitsu, Y. (1992). "Software lifetime and its evolution process over generations". *Proceedings of the Conference on Software Maintenance*, 63-69.

165) Teng, J., Jeong, S. & Grover, V. (1998). "Profiling successful reengineering projects". *Communications of the ACM* **41** (6), 96-102.

166) Tichy, W. (Ed.) (1995). "*Configuration Management*" (Trends in Software, no 2). John Wiley & Sons, 170 p.

167) Tran-Cao, D., Levesque, G. & Abran, A. (2002). "Measuring software functional size: towards an effective measurement of complexity". *Proceedings of the International Conference on Software Maintenance - 2002*, 370-376. IEEE Computer Soc.

168) Ulrich, W. (2002). "*Legacy Systems: Transformation Strategies*". Prentice Hall, 422 p.

169) Valenti, S. (2002). "*Successful Software Reengineering*". IRM Press, 300 p.

170) Visaggio, G. (2000). "Value-based decision model for renewal processes in software maintenance". *Annals of Software Engineering* **9**, 215-233.

171) Warren, I. (1999). *'The Reneissance of Legacy Systems: Method Support for Software-System Evolution*". Springer Verlag, 182 p.

172) Warren, I. & Ransom, J. (2002). "Renaissance: a method to support software system evolution". *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*. IEEE Computer Society.

173) Weiser, M (1982). "Programmers use slices when debugging". *Communications of the ACM* **25** (7), 446-452.

174) Yeh, D. & Jeng, J.-H. (2002). "An empirical study of the influence of deparmentalization and organizational position on software maintenance". *Journal of Software Maintenance and Evolution: Research and Practice* **14**, 65-82.

175) Yin, R. & Keller, R. (2002). "Program comprehension by visualization in contexts". *Proceedings of the International Conference on Software Maintenance*, 332-341. IEEE Computer Soc.

176) Zou, Y. & Kontogiannis, K. (2002). "Migration to object oriented platforms: a state transformation approach". *Proceedings of the International Conference on Software Maintenance - 2002*, 530-539. IEEE Computer Soc.