Fundamental Laws and Assumptions of Software Maintenance

Adam A. Porter
Department of Computer Science
University of Maryland
College Park, MD, 20742
aporter@cs.umd.edu

Abstract. *Researchers must pay far more attention to discovering and validating the principles that underlie software maintenance and evolution. This was one of the major conclusions reached during the International Workshop on Empirical Studies of Software Maintenance. This workshop, held in November 1996 in Monterey, California, brought together an international group of researchers to discuss the successes, challenges and open issues in software maintenance and evolution.*

This article documents the discussion of the subgroup on fundamental laws and assumptions of software maintenance. The participants of this group included researchers in software engineering, the behavioral sciences, information systems and statistics. Their main conclusion was that insufficient effort has been paid to synthesizing research conjectures into validated theories and that this problem has slowed progress in software maintenance. To help remedy this situation they made the following recommendations: (1) when we use empirical methods, an explicit goal should be to develop theories, (2) we should look to other disciplines for help where it is appropriate, and (3) our studies should use a wider range of empirical methods.

1. Introduction

All software systems evolve [1]. As they evolve they undergo numerous, successive changes -- to fix errors, improve performance or other attributes, and to adapt to new environments. The longer a system stays in service, the larger maintenance costs are, and, therefore, it's not surprising that maintenance costs often dominate initial development costs [2]. Clearly, improved tools, techniques, and processes can save a great deal of time and money throughout the software industry.

To realize these savings, researchers are studying maintenance from many perspectives.

Making individual modifications. To make changes developers have to understand the existing system, evaluate the effect of proposed changes, and then implement and validate it. To address these issues, researchers study topics such as design recovery, program comprehension, impact analysis and regression testing.

Coping with evolution. Because systems undergo many changes, not just one, other researchers focus on the nature of change and how to limit its impact. Research in this area includes structured programming, object-oriented programming languages, software architectures and configuration management.

Managing the maintenance process. Maintenance is costly and somewhat unpredictable. Thus, management tools for controlling maintenance are in great demand. Many researchers have developed metrics and models to predict when components are likely to need changing and how much that will cost.

These are only a few examples of maintenance research. There are many others. Clearly, software maintenance is a significant problem that continues to foster considerable activity. However, as with other software engineering areas, there is a concern that our research efforts lack hard evidence and critical evaluation, and that without these, we can't develop a deep understanding of what tools and processes work, when they work, or why. Consequently, many people believe that rigorous empirical methods must be one of the cornerstones of research in our field.

This belief brought a large group of researchers to the International Workshop on Empirical Studies of Software Maintenance. The workshop was help on November 8, 1996 in Monterey, California and involved researchers from software engineering, the behavioral sciences, information systems and management, and statistics. The goal of this meeting was to discuss the strengths, weaknesses, and open issues in empirical methods and determine how they could be profitably applied to improve software maintenance research.

The workshop attendees were divided into four groups.

1. Study and assessment of (new) technologies,
2. Methodologies for performing empirical studies,
3. Studies of the maintenance process: fundamental laws and assumptions, and
4. Maintenance process improvement in practice.

This article summarizes the findings of subgroup #3: *Studies of the maintenance process: fundamental laws and assumptions*. This subgroup is distinguished from others because their studies are not specific to particular tools or methods. Instead it tries to identify general principles. The group's participants have expertise in empirical methods, but approach their studies in different ways. Their backgrounds are diverse: software engineering, behavioral science, business, statistics, and others.

2. Summary of Discussion

Subgroup #3 had two working sessions. During the first session, each participant presented their research and took questions from the other participants. Before the workshop each participant received the following list of questions about their work.

1. What were the study's goal and questions?
2. What empirical methods did you use (e.g., controlled experiment, case study, survey)?
3. Describe your study's design.
4. What threats to validity did you consider (construct, internal, external)?
5. What did you do well and what will you change in the future?
6. How much did it cost to perform the study?
7. What new questions do your results raise?
8. Can other researchers replicate your study?

2.1 Synopsis of Presentations.

This section summarizes each presentation. Of course, the few paragraphs given to each presentation cannot capture the careful thought that goes into designing and conducting empirical research. Please see the complete proceedings. Any mistakes or misrepresentations are unintentional, and are the sole responsibility of the session chair, not the workshop participants.

- *Maintenance of Reuse-Based Domain-Specific Software Product Lines[3]*. Presented by William Thomas:

  Reuse saves money and time. Therefore, greater levels of reuse promise even greater savings. Although this seems to be correct, the authors urge us to look more closely. A common way to get high reuse levels is to split systems into two parts: one specific to the domain and the other to the application. The domain-specific part provides the infrastructure: interconnection support, common functions, and system architecture. Everything else is the application-specific part.

  The authors strongly agree that this approach reduces coding effort, but warn us that it may also redefine maintenance. Traditionally, applications are developed and maintained separately. However, when applications share code, they may need to be maintained as a unit. Of course, developers can still choose to maintain all applications individually, but may lose the benefit of domain-specific assets. They might also choose to maintain all related applications as a unit, but that may constrain individual applications.

  Since the best way to manage this situation is unclear, the authors will conduct empirical studies using historical data. They will explore such issues as measuring the cost of evolving applications and domains, coupling between applications and domain, patterns of change, and the effect of domain maturity on maintenance strategies.

  This work reminds us that new technology can challenge, even invalidate, our basic assumptions.

- *Using a Didactic Model to Measure Software Comprehension[4]*. Presented by Elisabeth Dusink

One of the first steps in changing a system is understanding it. Documentation is supposed to help understanding because it collects system knowledge and presents it in a structured way. The authors of this article claim that developers learn about a system by reading its documentation. Furthermore, they suspect that qualities of the learning process affect the quality of maintenance. Based on a model of learning by Bloom[5], they argued that learning progresses from gaining knowledge, to understanding it, applying it, analyzing it, and, finally, to synthesizing new knowledge from it.

The authors discussed a controlled experiment to link these levels of learning to maintenance success. The experimental subjects, second-year CS students, will be placed into three- and four-person groups. Each group will be given a 7,000 line email system written in C, and asked to modify it. They expect the change to take about 80 hours per person to complete. For each modification the experimenters will determine whether the group made the correct changes. Each person's understanding level will be measured via a questionnaire.

One potential outcome of this research is a low-cost index for maintenance success. That is, we may be able to test new technology by measuring its effect on understanding levels rather than by measuring its effect on maintenance tasks themselves.

- *Change-Prone Modules, Limited Resources and Maintenance[6]*. Presented by Warren Harrison

Maintainers often work under tight deadlines with meager resources. So sometimes they take short cuts even though they'd prefer to design and implement each change carefully. This kind of change is “ad hoc”. It can be made rapidly, but often degrades structure. In contrast “planned, structure preserving” changes may preserve structure, but are more expensive. Harrison argues that ad hoc changes aren't as damaging as has been claimed. He discussed data that showed that, in practice, changes tend to be confined to a small part of the system. His conclusion is that modules that rarely change cannot degrade and, thus, can tolerate ad hoc changes.

The author also outlined a model to help decide when to make ad hoc changes and when to restructure. In the model, the cost of a maintenance request is a function of the request’s complexity, the module’s state, and the kind of change made. Although the authors are still developing the model, it captures two important notions: (1) ad hoc patches degrade structure more than planned, structure preserving changes do, and (2) the effect of ad hoc changes will be compounded in modules that change often.

One interesting feature of this research is that it uses mathematical techniques to model long-term evolution. Since, by definition, evolution takes time, researchers must find ways to reason about it. Tools like these are a first step in that direction.

- *An Empirical Exploration of Code Decay[7]*. Presented by Adam Porter

Software systems must tolerate numerous, successive changes. As this happens, systems deteriorate and changes become increasingly difficult. Eventually, new functionality cannot be added and the system must be redesigned. Since solutions to this problem would be valuable, these researchers are conducting a long-term, multidisciplinary project to examine the fundamental causes, symptoms, and remedies for code decay.

The project team contains researchers in Statistics, Experimentation, Organizational Theory, Programming Languages, Software Engineering, and Visualization. Their primary data source is the AT&T 5ESS™ switching system and its development data. This includes the switch's source code and change control history (≈700,000 changes) covering the last 15 years, its planned and actual development milestones, effort and testing data, organizational history, development policies, and coding standards.

To help the project achieve its goals quickly, they have constructed a set of model code artifacts that they call the “code decay testbed”. To do this they developed several small systems, attempted to induce code decay in them, and used the resulting systems to evaluate ideas about code decay.

This research is a good example of how laboratory and field studies can work together to strengthen results.

- *On Increasing Our Knowledge of Large-Scale Software Comprehension[8]*. Presented by Anneliese von Mayrhauser

Far too often researchers create solutions (tools) without understanding the problem. Thus, there’s a tendency to focus on a tool’s novelty or performance rather than on its efficacy. The authors believe that program comprehension is a fundamental maintenance problem. Consequently, they have begun to study how software developers understand programs and how maintenance tools can make it easier.

They presented a case study in which they analyzed how 11 professional programmers performed maintenance. Then they tried to correlate what they saw with known theories of program comprehension. These theories suggest that several factors may affect maintenance: task type, prior exposure to the system, domain expertise, and language expertise.

The authors felt that limited sample size was one of the biggest problems that they and other experimenters face. Therefore, they argue that we will have to combine data from multiple studies. Further, this will require better methods for comparing studies, sharing data and terms, and analyzing aggregate data.

One implication of this is that experimental results must generalize beyond the environment from which they were taken. This may be one of the most important open issues in empirical software engineering.

- *The Software Maintenance Process in Portugal[9]*. Presented by Helena Mendes-Moreira

  Maintenance costs are the dues of success. In this work the authors surveyed 37 software development organizations in Portugal to characterize their maintenance processes. They selected companies with more than 500 developers and revenues in excess of 13 million dollars. After they examined the survey responses they conducted interviews with several of the respondents.

  They found that most of the maintenance performed in these organizations involved quick fixes, rather than planned enhancements. Also, in many cases, support activities such as updating documentation were not done. More importantly, they found that the ratio of maintenance effort to new development effort was growing rapidly. This might be expected in companies that enter the software market, develop new products, and now, for the first time, must maintain them. Several participants noted similar patterns in US industry.

  This work raises the possibility of comparing historical data from established organizations with current data from emerging ones. Would they find common patterns of organizational growth despite the changes in technology?

- *Predicting Aspects of Maintainability from Software Characteristics[10]*. Presented by Jarrett Rosenberg

  We have to change the way we study change. The author presented several studies that tried to predict the likelihood and cost of certain maintenance tasks. His results cast doubt on the soundness of current metric-based modeling approaches. First he found little correlation between static complexity metrics and repair activity. Second, he found considerable and unexplained variation in the metric values, suggesting that other, unknown, factors are driving maintenance.

  The author stressed two flaws in current metrics research. Metrics tend to be purely syntactic, which excludes vital semantic information. They also tend to be static, ignoring information about a system's development and evolution (e.g., testing history). The solution, however, is not to define new measures. Instead, he argued, we will need to focus on creating deep theories of maintenance-related factors.

  This presentation stressed two important research goals: to continue raising our scientific standards, and to borrow wisely from the ideas and approaches of other research areas.

- *Operational System Complexity[11]*. Presented by Scott Schneberger

  Maintenance bottlenecks can shift as technology changes. The industry is moving more and more from centralized to distributed architectures. Will this shift change our assumptions about maintenance? Schneberger suggests that it might. Since, he claims, there is little data on the maintenance of distributed applications, he performed an exploratory study.

  In this work he modeled a system's complexity in terms of its components, their internal complexity, and their interactions. To test this idea he conducted a survey. His initial results were that as distribution increased (i.e., components spread over more processors), system complexity increased, while component complexity decreased. One of his conclusions is that distribution shifts complexity out of individual components and into their interconnections.

  The author also points out that, in some domains, there is a trend back toward centralization. These results suggest that the benefits of solving certain maintenance problems can change over time.

- *The Impact of Documentation Availability on Software Maintenance Productivity[12]*. Presented by Eirik Tryggeseth

  One way to evaluate technology is to ask what would happen if it didn't exist. The author takes this approach to see how documentation affects maintenance success. He also asks whether this effect is different for more- or less-skilled programmers. To do this he performed a controlled experiment with 34 undergraduate students in computer science as subjects. Before the experiment he also measured the students' skill in reading and writing C++ programs.

During the experiment the participants were asked to modify a 2700 line C++ program. Half of them, group B, were given documentation, the rest, group A, were not. The experimenters measured the amount of time that the participants spent understanding and modifying the system, and they measured the quality of the modification. They found people who had documentation, group B, spent less time understanding the system and made higher quality changes. They also found that the performance measures for group B were correlated with the skill measures, while those of group A were not. One explanation might be that Group B understood the system quickly, so programming skill became the limiting factor for task performance. Group A, however, found it hard to understand the system, and, therefore, were unable to profit from their programming skill.

One implication of this result might be that system-level knowledge has a greater (or, at least, more immediate) effect on maintenance quality than individual programming skill does.

- *Assessing Maintenance Processes Through Controlled Experiment[13].* Presented by Guiseppe Visaggio

  Sometimes the simplest approach is the most cost-effective. In this work the author explores the cost-benefit tradeoffs between quick-fixes and a more thorough change process, called iterative enhancement. The goal of this work is similar to Harrison's (described earlier), but uses a controlled experiment rather than mathematical modeling.

  The participants in this experiment were asked to modify two different systems, once using the quick-fix approach, once using an iterative enhancement approach. The author measured the correctness, completeness, effort, and traceability of each change. They found that quick fixes were less reliable and degraded structure more than iterative enhancements did. One other interesting result was that quick-fixes were done faster only when modification affected fewer than ten modules.

  One fascinating observation is that these results appear to agree with those discussed by Harrison even though the researchers used very different research methods.

2.2 Organizing the Current Literature

In the second session the group analyzed their presentations and other existing research to see what, if any, fundamental laws of software maintenance are known. Because maintenance has so many facets, we first developed a very rough taxonomy of the factors that might affect maintenance. Then for each category we tried to synthesize common results, hoping to identify potential laws. We thought this might also indicate open areas that should be, but are not being, addressed. The categories represent factors related to product, people, process, and task.

1. **Product.** These factors relate to how attributes of system artifacts affect maintenance. We divided product factors into two subcategories: **complexity** and **structure**. We assumed that complexity affects our ability to understand the product. Research in this area includes software complexity metrics and structured programming. Structure refers to how system components are organized. We assumed that structure affects how changes impact a system. Research in this area includes studies of ripple-effect, development of design patterns, and studies of software architecture and domain-specific languages.
2. **People.** These factors relate to how the attributes of individuals and groups affect maintenance. We discussed three levels of this factor: Individual, Team, and Organizations. We assumed that individuals' abilities, group dynamics, and organizational constraints affect maintenance. Research in these areas include program comprehension, groupware, and cycle-time reduction studies.
3. **Process.** These factors relate to how the activities that individuals and teams carry out routinely affect maintenance. We assumed that process factors affect an organization's ability to predictably achieve their development goals. Research in this area includes documentation approaches, and configuration management tools.
4. **Task.** These factors relate to the viewpoint from which maintenance is studied. Some research focuses on individual changes while other focuses on longer-term evolution processes.

Within each of these categories, we tried to extract common findings. One topic for which we had some success was modularization. Several people stated that Parnas' early work on information hiding [14] illustrated some benefits of modularization, and that the effect of this and other research can be seen in today's object-oriented programming languages. See Kemerer [15] for a survey of results in this area.

Unfortunately, we were unable to find too many more. Certainly some exist. But they didn't spring to mind quickly. Although the group members knew of many studies, we couldn't distill their messages. Sometimes group members disagreed about the interpretation of single paper. Sometimes one paper's findings were in conflict with another paper's. Which one, if either, was correct? In many other cases we didn't think the work proposed any general findings.

Our interpretation is that as a field, we've asked many questions and taken many measurements, but what we've learned is unclear. Thus, our first conclusion is that we need to review the many studies published in the literature to synthesize potential theories.

2.3 The Next Steps

Given more time, we probably could have found more common results, but the difficulties we had were startling. One problem is that there hasn't been enough emphasis on synthesizing individual results into theories. But another, more fundamental, problem is that many studies aren't designed to produce general results.

Consider the following kinds of empirical research.

Feasibility studies. These studies are meant to validate new technology. Typically, the experimenter exercises his or her tool or method to show that it performs better than some other method or tool. These studies compare performances, but they rarely focus on the properties that make one tool better than another.

Statistical Modeling. Many researchers have modeled the relationships between various software metrics and maintenance attributes, such as change effort, severity, and locality. Since these models fit data without understanding it (correlation vs. causality), there's a very little reason to believe that they will apply to other data sets.

Observational studies. We also see many studies that document the behavior of a single project or organization. These studies may be useful as benchmarks of typical behavior, but they aren't intended to test any hypotheses. In fact, their authors rarely draw any actual conclusions..

Each of these types of studies serves a purpose and can further our understanding of software engineering. However, they are not designed to and are not likely to produce general theories. Thus, our second conclusion is that theory building must be designed into our empirical research.

3. Future Challenges:

In the previous sections we reviewed some of the subgroup's initial discussions. The group's activity took place in two sessions. In the first session, the participants presented their studies and discussed their goals, strengths, and weaknesses. During the second session we created a scheme for classifying maintenance studies, classified studies using this scheme, and then looked for common research findings in each class. We found some common findings, but not as many as we thought we should have. This led us to look more carefully at the field and ask why common results were so hard to find. Our main conclusion is that a critical part of scientific activity has been neglected in software maintenance research. That part is theory building – which can be done either by synthesizing individual results, or by proposing initial hypotheses and testing and refining them.

The last issue we wrestled with was how to remedy this situation. In this section we discuss some of our recommendations for dealing with this problem. We divided these recommendations into three groups: rethinking the goal of empirical methods, supporting interdisciplinary research, and expanding the use of empirical methods.

3.1 Rethinking the Goals of Empirical Methods

The quality of empirical research has improved tremendously in the last few decades. But it must continue to improve. In particular, we cannot forget that measurement is only one part of scientific inquiry. We routinely use measurement to describe, predict, and test. This is important, but, by itself, it does not give us the deep understanding we need to control software development.

- To gain control over software development we need to have validated theories that are (1) general, (2) causal, and (3) suggestive of control strategies.

  General theories hold across several environments. Relationships that only hold in one environment are still important (for instance, for process improvement), but as scientists we should not be satisfied with them. Thus, we should strive to draw theories from our studies (even if they turn out to be wrong) so that they can be tested in other environments.

  Theories should also be causal. Although the literature describes many reasonably-good predictive models, they only capture correlations. We shouldn't confuse correlations with the underlying principles we really care about. One of the key challenges to developing causal theories is to focus more on discovering underlying principles and less on measuring high-level performance.

Finally, if we have multiple candidate theories, we should prefer theories that suggest practical control strategies.

- Another issue to consider is human variation. Differences in natural ability affect every empirical study. Some researchers are looking at ways to account for these differences, but much more work needs to be done. As we discuss later, other fields have this problem as well and may have some insight that will help us.
- Although we are ultimately concerned with professional programmers who build industrial software, cost concerns lead us to use student subjects. Since we do not entirely understand the relationship between the student programmers and professionals, these studies are often discounted. Consequently, we need to develop models of the differences between student and professional populations.
- Science must be a public activity. There have been calls to make data public, but we must also share artifacts, procedures, and terminology as well. Repositories and web sites should be set up and greater efforts should be made to conduct collaborative research.
- Science is iterative. As we build and test theories, we will find almost all of them to be incorrect or imprecise. As authors and as reviewers, we will have to change how we think about and how we present our findings.

3.2 Supporting interdisciplinary research

Too often software engineering researchers think that our field is entirely unique. Software development involves a web of individuals, groups and organizations working to build a complex array of products. Therefore, it's certain that other fields have tools and techniques that we will find useful. For example,

- Behavioral scientists study how people work together (among other things). These have theories about how people learn and understand, and how they work together. One of the group members, Jarrett Rosenberg, coined the term “theory reuse” to describe this kind of interdisciplinary collaboration.
- Cognitive scientists study how people think. Consequently, they have experience defining instruments to measure aspects of human skill. As we discussed earlier, such instruments could play a large role in factoring human variation out of our empirical studies.
- Statistics already plays a big role in data analysis, experimental design, and hypothesis testing. We can also benefit from their knowledge of simulation methods, visualization methods, and mathematical modeling.
- Business disciplines such as organization theory and information systems management have a great deal of knowledge about how process, organizational structure, and business strategies affect people’s ability to get work done. Understanding these effects may help us to reconcile studies from multiple organizations. Also economists have considerable expertise in modeling complicated phenomena.

3.3 Expanding the Use of Empirical Methods

Controlled experiments are the standard method for verifying theories. However, they are not perfect. They are expensive, have limited external validity, and it may be very difficult or, even unethical, to use them in certain situations. Therefore, in addition to traditional analysis techniques, we should consider other methods for generating and testing theories.

- Qualitative analysis. Refers to the analysis of data that is represented in words and pictures rather than as numbers [16]. These approaches do not have as many supporting analyses as traditional quantitative methods do, but may provide a richer description of the phenomena being studied.
- Meta-analysis. Gathering enough data to draw sound conclusions is a major problem in empirical research. One way to solve this problem may be to integrate the data and results of multiple studies. This can be done on an ad hoc basis, but there also approaches for statistically integrating the data. These approaches are called meta-analysis.
- Mathematical modeling and simulation. This approach has been widely used in computer science, but not in software engineering. This is unfortunate because mathematical models can be powerful and much less expensive then experimenting with an actual system or process. They can also be very effective as a compliment to other approaches.
- Case studies. Many case studies are simply retrospective descriptions of a project. Sometimes we call these “lessons learned” articles or “experience reports”. We should try to do more case studies in which a hypotheses is stated and data is analyzed to see whether it is consistent with the hypothesis.

- Surveys. Surveys can be a very inexpensive way to acquire lots of information at low cost. They have problems, but again, there is a large community of researchers that use them and understand their limitations.

One final point to make here is that each of these approaches has strengths and weaknesses. Sometimes the best thing to do will be to combine two or more approaches.

4. Summary

Because software vendors must respond frequently and correctly to changing user expectations, hardware platforms, and alternative products, software systems must tolerate numerous, successive changes. In practice, however, systems deteriorate and changes become increasingly difficult to implement. Thus it's easy to see why the costs of maintenance often dominate the costs of initial development.

Research that makes maintenance easier will save considerable time and money throughout the software industry. Therefore, researchers have identified some sources of maintenance problems and developed potential remedies for it. For example, some researchers believe that repeated changes complicate a system's internal structure, so they have developed code metrics to model structural complexity. Others think that some software designs are inherently less flexible than others so they have focused on design patterns and software architecture. Still others argue that changes become more difficult because, over time, developers lose their understanding of the system They have developed reverse engineering tools, and formal documentation.

The exact manner and degree in which different factors affect maintenance is unclear. It is clear, however, that this information is vital if our research is to have sustained, predictable improvement. Because many researchers share this belief, they attended a workshop on the topic of empirical studies of software maintenance. This article tries to summarize the ideas of the subgroup on *Studies of the maintenance process: fundamental laws and assumptions*.

This group's main conclusion was that research community in software maintenance has produced many interesting results, but failed to consolidate them into useful theories. To help remedy this situation the group made three recommendations.

1. Researchers should design studies whose goal is to generate, test, and refine useful theories, not simply to describe behavior. This implies that we must use much more sophisticated empirical methods than we currently do.
2. Researchers should look to other disciplines for helpful insights, tools, and theories. However, they should borrow wisely.
3. Empirical research is severely limited by our inability to find adequately-sized samples. Researchers need to consider non-traditional techniques for combining data from multiple sources, and need to use multiple data collection and analysis approaches to make better use of small data sets.

5. List of participants

The session chair would like to thank the subgroup participants for helping to make the workshop a success: William Thomas, Helena Mendes-Moreira, Elisabeth Dusink, Warren Harrison, Jarrett Rosenberg, Giuseppe Visaggio, Marie Vans, Anneliese von Mayrhauser, Eirik Tryggeseth, Scott Schneberger.

6. Bibliography

1. Belady, L.A. and M.M. Lehman, *A Model of Large Program Development.* IBM Systems Journal, 1976. **15**(1): p. 225--252.

2. Gibson, V.R. and J.A. Senn, *System Structure and Software Maintenance.* Communications of the ACM, 1989. **32**(3): p. 347--358.

3. Thomas, W. and J. Baldo. *Maintenance of Reuse-Based Domain-Specific Software Product Lines*. in *International Workshop on Empirical Studies of Software Maintenance*. November 1996. Monterey, CA.

4. Dusink, E. and P.G. Kluit. *Using and Didactic Model to Measure Software Comprehension*. in *International Workshop on Empirical Studies of Software Maintenance*. November 1996. Monterey, CA.

5. Bloom, B., ed. *Taxonomy of Educational Objects. The Classification of Educational Goals. Handbook 1. Cognitive Domain*. . 1968, David McKay Company: New York.

6. Harrison, W. *Change-Prone Modules, Limited Resources, and Maintenance*. in *International Workshop on Empirical Studies of Software Maintenance*. November 1996. Monterey, CA.

7. Karr, A., A. Porter, and L. Votta. *An Empirical Exploration of Code Evolution*. in *International Workshop on Empirical Studies of Software Maintenance*. November 1996.

8. von Mayrhauser, A. and A.M. Vans. *On Increasing our Knowledge of Large-Scale Software Comprehension*. in *International Workshop on Empirical Studies of Software Maintenance*. November 1996. Monterey, CA.

9. Castro, M. J. H. Mendes-Moreira. *The Software Maintenance Process in Financial Organizations*. in *International Workshop on Empirical Studies of Software Maintenance*. November 1996.

10. Rosenberg, J. *Problems and Prospects in Quantifying Software Maintainability*. in *International Workshop on Empirical Studies of Software Maintenance*. November 1996. Monterey, CA.

11. Schneberger, S.L. *Position Paper for the International Workshop on Empirical Studies of Software Maintenance*. in *International Workshop on Empirical Studies of Software Maintenance*. November 1996. Monterey, CA.

12. Tryggeseth, E. *The Impact of Documentation Availability on Software Maintenance Productivity*. in *International Workshop on Empirical Studies of Software Maintenance*. November 1996. Monterey, CA.

13. Visaggio, G. *Assessing Maintenance Processes Through Controlled Experiment*. in *International Workshop on Empirical Studies of Software Maintenance*. November 1996. Monterey, CA.

14. Parnas, D.L., *On the Criteria for Decomposing Systems into Modules.* Communications of the ACM, 1972. **15**(12).

15. Kemerer, C.F., *Software Complexity and Software Maintenance: A Survey of Empirical Research.* Annals of Software Engineering, 1995. **1**: p. 1--22.

16. Gilgun, J.F., *Definitions, Methodologies, and Methods in Qualitative Family Research*, in *Qualitative Methods in Family Research*. 1992, Sage.