

UPGRADE is the European Journal for the Informatics Professional, published bimonthly at <<http://www.upgrade-cepis.org/>>

Publisher

UPGRADE is published on behalf of CEPIIS (Council of European Professional Informatics Societies, <<http://www.cepis.org/>>) by NOVÁTICA <<http://www.ati.es/novatica/>>, journal of the Spanish CEPIIS society ATI (Asociación de Técnicos de Informática <<http://www.ati.es/>>).

UPGRADE is also published in Spanish (full issue printed, some articles online) by NOVÁTICA, and in Italian (abstracts and some articles online) by the Italian CEPIIS society ALSI <<http://www.alsi.it/>> and the Italian IT portal Tecnoteca <<http://www.tecnoteca.it/>>.

UPGRADE was created in October 2000 by CEPIIS and was first published by NOVÁTICA and INFORMATIK/INFORMATIQUE, bimonthly journal of SVI/FSI (Swiss Federation of Professional Informatics Societies, <<http://www.svifsi.ch/>>).

Editorial Team

Chief Editor: Rafael Fernández Calvo, Spain <rfoalvo@ati.es>
Associate Editors:

• François Louis Nicolet, Switzerland, <nicolet@acm.org>
• Roberto Carniel, Italy, <carniel@dgt.uniud.it>

Editorial Board

Prof. Wolffried Stucky, CEPIIS President
Fernando Piera Gómez and
Rafael Fernández Calvo, ATI (Spain)
François Louis Nicolet, SI (Switzerland)
Roberto Carniel, ALSI – Tecnoteca (Italy)

English Editors: Mike Andersson, Richard Butchart, David Cash, Arthur Cook, Tracey Darch, Laura Davies, Nick Dunn, Rodney Fennemore, Hilary Green, Roger Harris, Michael Hird, Jim Holder, Alasdair MacLeod, Pat Moody, Adam David Moss, Phil Parkin, Brian Robson.

Cover page designed by Antonio Crespo Foix, © ATI 2003

Layout: Pascale Schürmann

E-mail addresses for editorial correspondence:
<nicolet@acm.org> and <rfoalvo@ati.es>

E-mail address for advertising correspondence:
<novatica@ati.es>

Upgrade Newsletter available at

<<http://www.upgrade-cepis.org/pages/editinfo.html#newsletter>>

Copyright

© NOVÁTICA 2003. All rights reserved. Abstracting is permitted with credit to the source. For copying, reprint, or republication permission, write to the editors.

The opinions expressed by the authors are their exclusive responsibility.

ISSN 1684-5285

Next issue (Oct. 2003):
“e-Learning – Borderless
Education”

Software Engineering – State of an Art

Guest Editor: Luis Fernández-Sanz

Joint issue with NOVÁTICA

2 Presentation: Software Engineering. A Dream Coming True? – Luis Fernández-Sanz

The guest editor presents the issue, that focuses on a really broad field like Software Engineering (SE) which has been driving the evolution of software development since the late sixties of the past century. The papers cover different areas of interest related to the application of engineering principles to software development and maintenance. As usual, a list of useful references is also included for those interested in knowing more about this subject.

5 Software Project Management. Adding Stakeholder Metrics to Agile Projects – Tom Gilb

In this article the author offers an analysis of the implications of the new agile methods in the field of software development.

10 Model-Driven Development and UML 2.0. The End of Programming as We Know It? – Morgan Björkander

This paper focuses on the idea of a truly model-driven software and analyses the influence that the new version of UML (Unified Modeling Language) is having on this process.

15 Component-Based Software Engineering – Alejandra Cechich and Mario Piattini-Velthuis

This paper studies the important role that components play in the field of Software Engineering.

21 An Overview of Software Quality – Margaret Ross

The author reviews important issues concerning quality in software development and also deals with the issues of users with disabilities and the influence of legislation regulating this aspect.

26 Lessons Learned in Software Process Improvement – José-Antonio Calvo-Manzano Villalón, Gonzalo Cuevas-Agustín, Tomás San Feliu-Gilabert, Antonio de Amescua-Seco, M^a Magdalena Arcilla-Cobián, and José-Antonio Cerrada-Somolinos

This paper describes the lessons learned by SOMEPRO, a Software Engineering R & D group in the Universidad Politécnica de Madrid, in more than ten software process improvement projects.

30 A New Method for Simultaneous Application of ISO/IEC 15504 and ISO 9001:2000 in Software SME's – Antònia Mas-Pichaco and Esperança Amengual-Alcover

The authors offer their view, based on practical experiences, of the always thorny problem of applying best software development practices to organizations where resources are especially limited.

37 Empirically-based Software Engineering – Martin Shepperd

This paper presents an overview of empirical Software Engineering and its implications for practitioners and researchers in four areas (object-orientation, inspections, formal specification and project failure factors.)

42 Software Engineering Professionalism – Luis Fernández-Sanz and María-José García-García

The aim of this paper is to provide a brief overview of what goes into making up our true perception of software engineers as specialised professionals within the field of Information Technologies.

47 Searching for the Holy Grail of Software Engineering – Robert L. Glass

In this article, the author defends eclecticism in development methods and the contribution that Software Engineering should make in this respect whenever the nature of a project demands flexible methods in order to be successful.

49 Free Software Engineering: A Field to Explore – Jesús M. González-Barahona and Gregorio Robles

This article analyses the existing points of contact between Software Engineering and the development of free software, and puts forward a few future lines of research in this respect.

Empirically-based Software Engineering

Martin Shepperd

This article gives a survey of activity in the field of empirically-based Software Engineering. It argues that this is an important area of research if practitioners are to make decisions on better evidence than mere subjective opinion. The article describes four areas where empirical data has improved our understanding of software technology. These are object-orientation, inspections, formal specification and project failure factors. The article concludes that empirical Software Engineering is likely to grow in importance but that there remain challenges, not least in assessing large scale processes and artifacts, in dealing with the human or creative aspects of processes and in overcoming the publication bias against ‘negative’ results.

Keywords: case study, empirical research, experiment, Software Engineering.

1 Introduction

The idea of viewing development and maintenance of software systems as a systematic engineering discipline was first mooted in the 1960s by Randall at a NATO workshop, thus Software Engineering was born. Subsequently there have been much progress and many of the things we now take for granted as part of modern software development are the direct consequence of Software Engineering research over the past 35 years. This includes requirements specification techniques, methods for modelling and reasoning about software architecture, modularisation and component reuse, object-oriented methods and programming languages, validation and verification techniques, software testing, project management methods, incremental and iterative development approaches and so the list goes on. However, amongst this list of what might be argued to be successes are many ideas that have not stood the test of time or were promoted as universal panaceas but with the benefit of hindsight have been seen to be of only marginal or local significance.

The pace of growth and rate of change within the software industry is nothing short of extraordinary. But this has brought about many problems. There is a large folklore of failed, late and over budget software projects coupled with many well publicised software related disasters. Consequently the demand for improvements considerably outstrips supply! The result can be a certain degree of credulity and a need to sort the wheat from the chaff, as it were. Moreover, it is not simply a matter of determining whether the new Software Engineering technique being advocated is good, bad or indifferent, but, more to the point, determining under what *circumstances* the technique will be good, bad or indifferent. The mere fact that software developer X uses a new technique successfully does not guarantee that software developer Y will meet with equal success. X may have different staff recruitment and training policies, develop different products to different quality requirements, have different management practices and possess a

different culture to Y. For this reason systematic, empirically based research is essential in order to move Software Engineering on from an advocacy based discipline to an evidence based discipline.

The remainder of this paper briefly reviews the development and evolution of empirically-based Software Engineering. This is followed by a description of four areas of activity that might be characterised as technology evaluation. These are object-orientation, inspections, formal specification and software project failure factors. The paper concludes by summarising those areas where good progress has been made and by considering those topics that need to be addressed in the future.

2 A Brief History

Probably one of the earliest published studies that adopted an empirical and quantitative view of a software system was by Benington [1] based on his experiences with the SAGE project, one of the first large scale software projects. The paper contains many observations with which we would be familiar, but also detailed empirical data on software development costs, including distribution of costs by phase. This allowed the insight that testing consumed a considerable proportion of total costs, and of the order of five times more than coding.

The two main areas of interest of the 1960s and 1970s were tracking and predicting overall project costs and very fine grained work looking at code ‘complexity’ metrics (e.g. Halstead’s Software Science and McCabe’s cyclometric complexity metric). The cost models typically were size based with many modifiers (or cost drivers). This formed the basis of

Dr Martin Shepperd is Professor of Software Engineering at Bournemouth University in the UK. His research interests include empirical Software Engineering and the use of machine learning methods to build prediction systems. He has published more than 80 papers and three books. He is co-Editor in Chief of the journal Information & Software Technology and Associate-Editor of IEEE Transactions on Software Engineering.
<mshpepper@bournemouth.ac.uk>

well known models such as COCOMO in the early 1980s. Very much in contrast, the ‘complexity’ metrics were concerned with small fragments of code and much of the work focused on the minutiae of different programming language syntax and coding structures. Whilst, with hindsight, this work has not been perceived as particularly fruitful, but nevertheless allowed ideas of empirical validation to evolve and acceptance of the proposition that it is insufficient for researchers merely to propose new metrics or ideas without empirical support. It has also been the basis for much refinement and development of empirical methods suitable for the specific challenges of studying Software Engineering.

The 1980s and 1990s were characterised by the growth of, and broadening, interest in empirical Software Engineering. There was much diversification so that products other than code (e.g. designs, specifications, test plans, user guides) were also the subject of empirical study. Likewise interest extended beyond products to embrace processes e.g. designing, debugging, testing, comprehending and so forth. There was also more emphasis on multi-disciplinarity and explicitly drawing upon other areas of expertise such as statistics, psychology, management science, economics and measurement theory.

An important recent development is the growing interest in meta-analysis. That is how do we combine the results from several empirical studies? This can help address two situations. First, how much more confidence can we have in a result if it’s reported by several independent studies and second, how do we deal with the situation where not all results are in agreement? Vote counting is a simple but not always the most effective technique. Other techniques are being proposed for combining results in order to summarise results more generally within an area of enquiry (see for example, [2]). The main challenges are to ensure that the data is relatively homogenous, for examples that results are not influenced by cultural differences if studies are conducted in different countries, or where such differences exist, they are well understood so that the population can be appropriately partitioned.

3 Some Case Studies Evaluating Technology

In this section, we briefly the results from four areas of Software Engineering that have attracted considerable interest and debate over the years.

3.1 Object Orientation

Although the original ideas behind object oriented technology (OOT), derive from work on the programming language Simula in the 1960s, it was not until the 1980s that the work was popularised and its use became more widespread. Presently, C++ and Java are widely used and widely taught. The OO paradigm could be regarded as the orthodoxy of the late 1990s onwards. However, we have comparatively little empirically based knowledge of the behaviour of systems that have been implemented using OOT. Thus as OOT, continues to be heavily invested in, the need for research into better understanding, and prediction, of the behaviour of OOT is growing.

An example investigation based at a large European telecommunications company was conducted by Cartwright and Shepperd [3] which revealed a lot of useful information, particularly regarding the distribution of defects within the software. The organisation employed approximately 20,000 staff and more than 2,000 software developers. A disciplined approach was adopted for software design and implementation and the company was ISO 9000 accredited. There was considerable emphasis upon software quality, in particular eliminating defects. As much as 45% of resources were devoted to testing and simulation. The system studied was a sub-system of a much larger industrial real-time telecommunications system which comprised several million lines of code (LOC) which had been evolving over the past ten years. Its success was central to the organisation’s financial health. The sub-system was written in C++ and had been designed using the Shlaer-Mellor method.

From the analysis a total of 259 unique defects were identified and traced back to the relevant locations within the software. This suggests an overall defect density of 1.94 KLOC-1 which compares quite favourably with defect levels quoted by Hatton [4] of 2.9 KLOC-1. Indeed, the figure is quite conservative since some of the defects that have been recorded were found after integration testing but prior to release. However, apart from providing some kind of benchmark this type of analysis can yield a number of practical lessons. For example, the data indicated a small number of very defect prone classes, and indeed a mere 22% accounted for 75% of all defects, more evidence of a 20:80 rule as reported by others, for example [5] and [6]. This suggests that it is not optimal to distribute software verification effort evenly across the system. Instead there exists the potential for large productivity gains if the problematic components can be identified in advance since verification activity can be more effectively targeted.

This leads to the second useful result of this empirical study. It is possible to predict problematic components using information available at design time such as the number of events or states.

Third, the defects seemed to cluster around particular architectural features, most notably inheritance structures. Classes that were part of inheritance structures had a defect density more than three times greater than singleton classes (3.01 defects KLOC-1 compared with 0.90 defects KLOC-1). This was a particularly interesting finding – and confirmed by a number of laboratory experiments, e.g. [8] and [7] and other case studies e.g. [4] – since it suggested that this particular architectural mechanism provided by the object-oriented paradigm could, at the very least, be abused. This was despite the fact that at the time many pundits were promoting inheritance as the central part of OOT. Subsequently, it has become widely accepted that other mechanisms such as object composition are equally important and that inheritance can be an inappropriate mechanism particularly in circumstances where an object may change role. Thus we see empirical Software Engineering as an important adjunct to the development, deployment and evolution of software design methods.

3.2 Inspections

Another area that has attracted much interest is the use of inspections, particularly at design time, as an efficient technique for the early detection and removal of defects. Basically this involves reading or desk checking of artifacts e.g. design documentation, code, etc. by independent staff. This is typically followed by a meeting in which detected defects are assessed and, if accepted as valid, documented. See [9] for a good overview. However, whilst there is a consensus that inspections are an important technique there is rather more discussion as to how these should be conducted, for example, what is the optimum number of inspectors, is a face to face meeting essential, and so on? These are again examples of Software Engineering issues that can effectively be addressed by empirical analysis.

In the past few years there have been a number of experiments and studies. A good, recent example is that by [10] who present the results of a controlled experiment using 169 student subjects reviewing a 35 page requirements specification document that had been reseeded with 86 defects. They found that teams that used a mixture of reading techniques outperformed those that used a single technique. Another finding of some significance is the difference in performance between individuals and therefore the need to identify the more inefficient readers so that staff with the appropriate skills can be matched to right tasks. They also derived empirically-based models, in terms of cost-benefit, to determine optimum team size and inspection effort due to diminishing returns at the margin, i.e. the n th inspector or hour, in general yields more benefit than the n th plus one inspector or hour. Their analysis also classified defects by type. They reported that if one is primarily concerned with major defects then this different reading techniques are favoured than if the goal is to detect all defects.

Obviously one reservation is the use of student subjects, and indeed this a recurring problem for empirically-based Software Engineering. Generally it's not possible to use professionals for experiments for reasons of cost and access, consequently much published experimental work is based on the use of students. Some authors have argued that differences between student and professional subject may be less than might be expected [11]. More compelling is the use of industrially based case-studies to complement laboratory experiments.

3.3 Formal Specification

Formal specification is an example of an area of Software Engineering that has been strongly advocated by a sizeable part of the Software Engineering community. Clearly the idea of using a formal notation to describe and prove properties of a software specification is very attractive, particularly for applications that might be considered safety critical. Alternatively would time and effort be better devoted to developing n versions in a way that is analogous to using hardware redundancy to protect against failure? Persuasive arguments can be made for both points of view. Again empirical studies can shed light on this debate although it must be noted that formal specification embraces a very wide spectrum of activity from formal proofs of each stage and refinement to use of mathematical based notations in a 'lightweight' fashion. Moreover, no-

tations vary considerably from the model based approaches that are essentially set-theoretic coupled with first order predicate calculus, such as B and VDM, to algebraic methods such as OBJ and LARCH, to process algebras such as CSS.

An interesting, and not entirely positive, case study was published by [12]. Here the authors discuss their experiences of using Z (a model based formal specification language) to develop a parser. This was not very successful. The specification was found to be poor containing many errors and having problems with modularity. Although the final code defect density was quite good (1.3 faults per KLOC), many of the faults were serious and the reliability (as opposed to the correctness) was unacceptably poor. The use of Z was therefore regarded as unsuccessful since the emphasis was on an inappropriate software quality factor, i.e. correctness rather than reliability.

Another, and well publicised case study also based around Z, was conducted jointly by IBM Hursley and Oxford University [13]. This project applied formal methods (in practice primarily specification with limited amount of refinement or program proving) to an upgrade of the CICS transaction processing software system. The size of the upgrade for which Z was used amounted to approximately 48 KLOC which represented approximately 18% of the modified code (268 KLOC) and a total release size of 768 KLOC. The project team claimed that 19 defects were avoided as a result of using formal methods, i.e. a more than 50% reduction. They also claimed a cost savings of 9% of the total budget. This sounds very impressive and the case study has been widely cited to argue the merits of this approach to software development, see for example [14].

Unfortunately, more careful re-examination of the CICS data by Finney and Fenton [15] reveal major problems with this analysis. Whilst the researchers at IBM and Oxford University are to be commended in conducting an industrially based study there are many difficulties with their data, not least in determining what would have happened if Z had not been used, in other words there is no control. Also defect data is problematic since there is a time dimension: faults over what time period and under what circumstances? Moreover were the staff involved with the Z specified parts of the system representative of all IBM staff – there are grounds for thinking not – and were the Z specified parts representative of all the system? This is not to decry formal methods nor the efforts of those published these results but merely to point out that not all quantitative data is of equal value and care is needed in interpretation. A good review of methodological issues in conducting Software Engineering case studies is given by Kitchenham et al. [16].

3.4 Software Project Failure Factors

As mentioned in the introduction, despite the considerable progress that is being made by the software industry, there are still many examples of projects failing, sometimes in both dramatic and costly ways. Empirically-based Software Engineering has tackled this problem in two ways. First in terms of forensic analysis to determine what specific and often local lessons might be learnt. Good examples, of this type of work concern the problems with the London Ambulance Dispatch System and flight 501 of the Ariane V rocket of the European

Space Agency. Second are the more general analyses, often using survey methods, to identify failure factors. In either case, the purpose of such work is to replace speculation with evidential reasoning. We consider each approach in a little more detail.

The London ambulance service computer-aided despatch system (LASCAD) project is well a documented example of a software project failure that tragically resulted in the loss of life due to the major delays in ambulance arrival times [17] [18]. As a result a number of lessons were derived from this failure. Interestingly, many of these related more to the project management and less to technical issues despite the memory leak that caused the system to slowly grind to a halt as computer memory was acquired but then not released. Examples of managerial problems include the procurement process itself which led to the contract being awarded to a company that was substantially cheaper than any of its competitors. This low bid was indicative of an organisation that failed to properly understand the problem domain due to inexperience, rather than an organisation that was more efficient than its rivals. Another problem area was the failure to properly involve users (the ambulance staff) leading to poor requirements capture and hostility once the system was introduced. Dispassionate analysis of projects is frequently a rich source of data and insight into Software Engineering.

Another example of a software related failure was flight 101 of the Ariane V rocket, 37 seconds after launch when an integer overflow caused maximum deflection of the main boosters and the flight to be correctly aborted. The payload was valued at approximately 500m. Again this was documented and a detailed *post mortem* published by the European Space Agency [19]. There were many lessons to be learned, but perhaps the most striking two were that due to complex relationships between the many sub-contractors and the ESA, the software was not properly tested. In this case if the code had been executed when presented with appropriate flight data the code was guaranteed to fail! Second, the system was rightly seen as mission critical so that redundancy was built in as a safeguard against failure of an individual component. Hence there were two parallel inertial reference systems (where the failure originated). Unfortunately both were running identical code so both simultaneously failed. The lesson is that hardware notions of redundancy do not directly translate to software.

Other researchers have adopted a quite different approach by studying software projects in general rather than looking at the specific details of an individual case. Typically researchers sample the wider population of software projects by using techniques such as surveys. An interesting recent example is the work by Procaccino et al. [20]. Here they sampled 21 software professionals from a large North American financial organisation and used a mixture of structured interviews and questionnaires to obtain information on project success and failure factors with a particular emphasis on those factors that might be determined early on within the project lifetime. This yielded data on a total of 42 projects. One striking result is the difficulty in determining exactly what constitutes success or failure. In this study management considered 78% of the projects to have

been successful, whilst developers only rated 49% as successful. It would also have been interesting to obtain the users' views on this matter! This illustrates at least one challenge in conducting studies such as this since, namely, varying perceptions. Nonetheless, a number of factors emerged from their analysis, including the importance of user involvement but large numbers of users frequently created "*as much conflict as consensus*". Another factor was the presence of a project sponsor who was 'visibly committed' to the project. Project size was also found to be influential, with large projects being more likely to result in failure. Whilst this study was based only a single organisation similar results have been reported by other studies thus increasing our confidence in the findings, subject to the caveat that we need to understand the context within which such results have been derived. The Procaccino et al. study was carried out in a large organisation. We should be cautious about generalising to, say, small organisations that develop game software.

4 Summary and Future Directions

It has been argued that empirically-based Software Engineering is essential if the discipline and practice is to progress beyond mere opinion. Without properly conducted and independent empirical validation it is difficult to judge whether a particular technique or software tool is valuable or otherwise. The situation is compounded by the fact that many commentators have a vested interest. If I wish to obtain research funding or consultancy for, say, software testing then I may over-emphasise the problems associated with not testing properly, and so on. Anecdotal evidence may be of some value, however the difficulty can be in properly understanding the context so as to allow reasonable generalisation to other situations. Empirical investigation therefore should act as a gatekeeper to new ideas within Software Engineering.

Whilst much progress has been made over the years, there naturally remain several of challenges.

First, there is scalability. Many of the problems of Software Engineering are associated with the scale of the tasks and artifacts that undertaken. Unfortunately this cannot always be adequately replicated in the laboratory. However, access to industrial sites is often restricted by either confidentiality or cost considerations. Better collaboration between researchers and industry will be invaluable.

Second, we are dealing with humans and an essentially creative or design activity. This introduces an enormous amount of variability into Software Engineering processes, that without extensive replication, can confound what may be small effect sizes derived from the phenomenon of interest.

Third, we need to cope better with the publication bias that favours 'positive' rather than 'negative' results. This can impact meta-analysis if not all 'negative' results are available in the public domain. It can also put pressure on researchers to 'trawl' for results or subconsciously inflate the evidence for claims made.

Notwithstanding these challenges, the field is of growing importance and has a strong future in supporting the software industry in making cost effective decisions.

5 Resources

The following section provides some starting points for readers interested in discovering more about empirically-based Software Engineering.

There are a number of good books, however, the best known are probably “Software Metrics: A Rigorous and Practical Approach” by Norman Fenton and Shari Pfleeger [21] and a more recent work by Claes Wohlin et al. entitled “Experimentation in Software Engineering: An Introduction” [22] which deals with many of the practical and methodological issues of carrying out this type of research. A number of journals also publish relevant articles. *Empirical Software Engineering*: an international journal is dedicated to this theme, however, many of journals contain much useful material including: *IEEE Transactions on Software Engineering*, the *Journal of Systems & Software* and *Information & Software Technology*. In addition magazines such as *IEEE Software* frequently contain shorter and more practitioner oriented articles.

There are also a number of annual conferences devoted to empirically-based Software Engineering. The most important are the *IEEE International Metrics Symposium* and the *Empirical Assessment and Evaluation in Software Engineering (EASE)* workshops.

There are also many good resources on the web. The International Software Engineering Research Network (ISERN) is an international network of many of the main research groups who are actively involved in empirically-based Software Engineering research. The home page is <<http://www.iese.fhg.de/isern/>>, which contains information about various ongoing experiments, technical reports that can be downloaded, etc.

Other research groups who maintain websites include:

- Experimental Software Engineering Group at the University of Maryland (USA):
<<http://www.cs.umd.edu/projects/SoftEng/tame/>>.
- Empirical Software Engineering Research Group at Bournemouth University (UK):
<<http://dec.bournemouth.ac.uk/ESERG/>>.

References

- [1] H. D. Benington Production of large computer programs, Symp. on Advanced Computer Programs for Digital Computers, Washington, D.C., Office of Naval Research, 1956.
- [2] W. Hayes. Research synthesis in Software Engineering: a case for meta-analysis, 6th IEEE International Softw. Metrics Symp., Boca Raton, FL: IEEE Computer Society, 1999.
- [3] M. Cartwright and M. J. Shepperd. An Empirical Investigation of an Object-Oriented Software System. *IEEE Trans. on Softw. Eng.*, 26(8), pp786–796, 2000.
- [4] L. Hatton. Does OO Sync with How We Think, *IEEE Software*, 15(3), pp48–54, 1998.
- [5] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object-Oriented Design, *IEEE Trans. on Softw. Eng.*, 20(6), pp476–493, 1994.
- [6] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity, Object-Oriented Series*, Prentice Hall: Englewood Cliffs, New Jersey, 1996.
- [7] J. Daly et al. Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software, *Empirical Software Engineering*, 1(2), pp109–132, 1996.
- [8] B. Unger and L. Prechelt. The impact of inheritance depth on maintenance tasks – Detailed description and evaluation of two experimental replications, Technical Report, Dept. of Comp. Sci., Karlsruhe University, Germany, 1998.
- [9] O. Laitenberger and J.-M. DeBaud. An encompassing life cycle centric survey of software inspection, *J. of Systems & Software* 50(1), pp5–31, 2000.
- [10] S. Biffl and M. Halling. Investigating the defect detection effectiveness, and cost benefit of nominal inspection teams, *IEEE Transactions on Software Engineering*, 29(5), pp385–397, 2003.
- [11] M. Höst et al. Using students as subjects – a comparative study of students and professionals in lead-time impact assessment, *Empirical Software Engineering* 5, pp201–214, 2000.
- [12] M. Neil et al. Lessons learned from using Z to specify a software tool, *IEEE Transactions on Software Engineering* 24(1) pp15–23, 1998.
- [13] I. Houston and S. King. CICS Project Report – Experiences and Results from the Use of Z in IBM, in *Lect. Notes Comput. Sci.* Vol. 551, Prehn, S. and Toetenel, W.J., Editors, Springer-Verlag: New York. pp588–596, 1991.
- [14] J. P. Bowen and M.G. Hinchey. 7 More Myths of Formal Methods, *IEEE Software* 12(4), pp34–41, 1995.
- [15] K. Finney and N.E. Fenton. Evaluating the effectiveness of Z: The claims made about CICS and where we go from here, *J. of Systems & Software* 35(3), pp209–216, 1996.
- [16] B. Kitchenham et al. Case-Studies for Method and Tool Evaluation, *IEEE Software* 12(4), pp52–62, 1995.
- [17] M. Hougham. London Ambulance Service computer aided dispatch system, *Intl. J. of Proj. Mngt.* 14(2) pp103–110, 1996.
- [18] P. Beynon-Davies. Human error and information systems failure: the case of the London ambulance service computer-aided dispatch system project, *Interacting with Computers* 11(6): pp699–720, 1999.
- [19] J. Lions. Report of the Inquiry Board for Ariane V Flight 501 Failure, Joint Communication ESA-CNES, Paris, France, 1996. (Also available from http://www.esa.int/export/esaLA/Pr_33_1996_p_EN.html).
- [20] J. D. Procaccino et al. Case study: factors for early prediction of software development success, *Information & Software Technology* 44(1), pp53–62, 2002.
- [21] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*, London: International Thompson Publishing, 1997.
- [22] C. Wohlin et al. *Experimentation in Software Engineering: An Introduction*, Norwell, MA: Kluwer Academic, 2000.