Proceedings of the 16th IRIS.
J.P. Bansler, K. Bødker, F. Kensing,
J. Nørbjerg, and J. Pries-Heje (Eds.).
Department of Computer Science,
University of Copenhagen, 1993

Embedded Software: Industry versus Research

Søren Lauesen, Jan Pries-Heje & Bodil Schrøder
Copenhagen Business School
Howitzvej 60, DK-2000 Frederiksberg
e-mail: sl.iio @ cbs.dk

Abstract

This paper summarizes the results from a Danish study of development methods used for embedded software. It compares the needs of industry with the apparent lack of influence from Danish research.

The main conclusions from the study are: (1) Industry have many problems related to development of embedded software, for instance ensuring that essential issues are dealt with early and not detected at the end of testing. (2) Only very few research results are transferred to industry. The lack of technology transfer is primarily caused by the researchers' lack of appreciation of the real problems in industry. (3) Even very good and very relevant results usually fail to be taken into regular use in industry. Many different barriers cause this failure.

Finally the paper suggests how these technology transfer problems could be overcome.

1. Introduction

In the Danish electronic equipment industry, software development is becoming more and more important. Often more than 50% of the development costs are used for software development. Therefore, improved methods are essential, and research could be a source for that.

This paper reports some of the results from a Danish interview study conducted during the winter 1992-93. Fifty persons from 17 Danish organisations were interviewed using a structured interview guide. Each interview, with one or more persons, lasted about three hours. The 17 organisations were carefully chosen such that all major Danish computer science departments

(DTH, DIKU, DAIMI, AUC), all major intermediary organisations (The Danish Technological Institute, ElektronikCentralen, Institute of Applied Computer Science (IFAD), private consultants), and major parts of the electronic equipment industry (telecommunication, electronics, measurement) were represented.

Each interview involved two or three researchers. One researcher conducted the structured interview, while the others directly observed the interview and recorded everything. Furthermore every interview was tape-recorded to support the written records. In addition, we occasionally studied selected development documents.

All the written records from the interviews were then analyzed using analytic induction [Miles & Huberman, 1984], and a full report of the results was produced [Schrøder et al., 1993]. In addition we have drawn on our own working experience from many projects in several companies.

2. Problems in practice

A major part of our observations was about the problems companies experienced in practice. Below we have categorized the main problems according to the typical phases of software development. A general conclusion is that most of the problems occur in the initial phases, but they show up in the late testing phases, where they are very expensive to repair.

2.1. Product definition phase

Defining the product concept and the requirements for new systems was a problem in all the companies we interviewed. We found no systematic way to link customer needs to the specification of requirements and further on to estimates for development time and cost.

The typical requirements specification we saw was unprecise and lacked important requirements such as performance, portability and maintainability. Furthermore, the requirements specification was often written in such a way that the marketing department and/or the customer could not fully understand it. As a result, important requirements were discovered late in the development process, making them very expensive to fulfill.

Also, implicit requirements like cost and deadline for delivery were not written down explicitly. At the same time, product development is so risky and so expensive that new products are seldom initiated. So when a new product is initiated, it has to provide a lot of new functionality. Together these two things cause a high risk. The obvious solution is to prioritize requirements instead of making a shopping list of requirements, but such priorities are not established. In the end, the project runs short of time and resour-

ces, and emergency priorities have to be used, but in an unplanned manner.

2.2. Design phase

Early in the system design, hardware parts and software parts are defined. This definition is seldom done systematically. Usually, all the vague requirements end up in the software part - a major reason for the huge schedule delays caused by software development. This also gives software developers a bad reputation compared to hardware developers.

The software design process is often not documented. Major parts of the design exists only in the heads of the developers. Typically, the input and output of each module is specified, but not how they relate. Even if the design is documented, for instance as dataflow diagrams or module specifications, the rationale behind the design decisions cannot be found in the documents. Furthermore it is impossible to trace the design to requirements, thus making it difficult to check whether the design is complete or whether parts of the design are superfluous.

Often real-time issues like resource sharing, deadlocks, or performance bottlenecks are only considered for single modules, but not for the system as a whole. One reason is that very few persons in the project can overlook the entire system. The result is that the problems turn up in integration testing or later, and they are extremely difficult to repair then.

We have not seen anyone trying to estimate performance using simple mathematics or simulation. The result of this lack of insight into where performance problems may occur is that the problems are detected late, so that only local optimization can be used to improve the situation.

User interface design is also causing problems. In the closed system parts (typically using dedicated hardware), the technical designers also take care of the user interface. They often know how to make prototypes and involve users in the design process, but they rarely do it in practice. In the open system parts (with screens and keyboard), the companies seem to rely heavily on Graphical User Interfaces, hoping that the use of standardized windows, icons, menus etc. automatically will lead to increased usability. It is well known that this is no guarantee for usability, but user involvement and user testing is needed too. Again, this rarely takes place.

2.3. Programming phase

The programming phase has fewer problems than the other phases. The programming language C dominates completely and has even wiped out most Assembler programming. Many companies are looking into C++ and object oriented programming, but we did not find any completed projects based on

C++.

The most serious problem we found was a lack of traceability, this time from design to program. For instance, we found cases where the program modules did not at all look like the modules from the design: In effect, all the design decisions had been remade.

2.4. Testing phase

The different kinds of testing (module, integration, and acceptance testing) take too much time, because the problems from the earlier phases show up now. Things missing in the requirements specification appear now: The performance is not good enough. Important customer needs are not covered. The dedicated hardware parts of the system do not work as expected.

All these problems come on top of the deadline problem. This means that the project is running short of time and resources exactly at the time when these serious problems are identified.

2.5. Maintenance phase

Maintenance seems to be plagued with a lot of problems. Old systems are often quite chaotic because: (1) The people who made the system have left. (2) The documentation is not up-to-date, or was never made. (3) The original design has been violated again and again, for instance with special modifications for important customers. (4) The hardware platform on which the product was based is dead, so that you cannot get spare parts, and testing can only take place at customer sites. (5) Earlier maintenance was often done in panic by people without sufficient knowledge of the entire system.

In recognition of the maintenance problems, some companies now put more emphasis on documentation. However, it is rarely done while development takes place, but attempted after delivery of the final system. (When other tasks become urgent, the documentation is stopped, of course). For instance, we have seen cases where the project manager claimed that they followed a certain method with full documentation of the design. When we asked to see the design document, we got the answer that it was not made yet, but when they had finished programming, they would do it!

Behind the documentation problem is a more basic problem: It is unclear what documentation would be useful. We saw several meters of documentation collecting dust on the shelves - it was simply not useful. The only documentation that always seems to work is thorough comments in the actual code.

3. Lack of relevant research to transfer

A lot of software related research is taking place in Denmark. Unfortunately, most of the research seems to have little relevance to the electronic equipment industry. This lack of relevant software research seems to be directly linked to the researchers' lack of industrial knowledge, which seems to be caused mainly by the position of the researcher:

(1) A researcher earns his merits primarily from publishing in international journals. That means that the researcher has to come up with something new and publish it to other researchers. Industrial relevance is seldom taken into account. Searching for good solutions to industry problems is of little interest to the researcher, because there is no credit for applying research results to industry.

(2) A researcher's environment consists of other researchers, possibly supplemented by sporadic industry contact. Furthermore, a researcher often has no experience other than being a researcher. This leads to fatal misjudgements. For instance we found researchers who believed that they knew what is going on in industry. But from our interviews we could see that their beliefes were based on prejudices.

(3) A researcher can indulge in a very limited problem area and dig very deeply into it. Most industrial problems involve solving many interrelated problems at the same time. So a narrow solution, no matter how good it is, will often fall short of the needs in industry.

4. Three examples of relevant research

During our interviews, we encountered two very promising research projects: Mjølner and KAITS. Despite their promises, both projects failed to gain regular use in the industry due to a number of reasons.

We also encountered a methodology called SPU, which seems to have gained wide-spread use in the Danish electronic equipment industry. The methodology was not invented by Danish researchers but was constructed by four practitioners based primarily on their own experiences.

4.1. Mjølner

Mjølner was developed by researchers at DAIMI (University of Århus). The Mjølner BETA system is a software development environment supporting object oriented programming in the BETA programming language. Mjølner seemed to be a very good solution for at least two important problems in the electronic equipment industry: Traceability and maintenance of design docu-

ments. However, Mjølner has not been well received. Our interviews point to two main reasons for that:

(1) Risk aversion. Companies were afraid of trying something new. The advantages seemed far too doubtful and long-termed compared to the expenses. Furthermore, you become dependent on a small development team for support. A key point is that companies doubted that the BETA language could be supported in the future on new processors.

(2) The new techniques were not compatible with existing techniques and tools. This created a special kind of risk. Customers could not return to the old techniques without losing all work based on the new tool. In contrast, C++ is acceptable for object oriented design, because you can always return to plain C again. But Mjølner forces you to change everything in your development process and gives you no easy way to return.

4.2. KAITS

KAITS was made as a cooperation between DTH (the Danish Technical University) and a large Danish company that sells measurement instruments worldwide. KAITS is a method developed especially for the design of real-time systems. KAITS provides a technique and a notation for identifying parallel processes, thereby deriving the task structure in a systematic manner. Today, KAITS is marketed as a tool called CEDER or CEDAR [Elmstrøm, 1989; Rischel et. al., 1987].

KAITS seemed to be a good solution for some common industry problems: (a) It guarantees optimal concurrency. (b) It allows the calculation of performance in advance. (c) It features very compact documentation which is easy to maintain. (d) It provides good traceability from design to program. However, KAITS was not transferred due to a number of reasons:

(1) The method was probably oversold as a solution for all problems in industry, even though it only supported a small part of the design process.

(2) Several companies heard about KAITS, but hesitated to use it until other companies demonstrated its success. This hesitation was also linked to the fact that industry products typically have a 10-year life cycle. And a product is typically bound to the same methods and tools as long as it is maintained.

(3) KAITS has a high learning threshold. Therefore it was considered too

risky to use.

(4) Practitioners were unable to read complex scientific papers on new techniques, in particular papers with mathematical notations as found in KAITS.

(5) In its present state, KAITS has weaknesses handling complex data structures.

4.3. SPU

In 1985 four practitioners wrote a book based on their experiences called Handbook on Structured Program Development, in short SPU [Biering-Sørensen et. al., 1988]. The SPU method includes all phases of program development: Requirements specification, design, coding, test and maintenance. In addition, SPU covers more traversing issues like project management, review, configuration management, and documentation.

The four practitioners inventing the method collected what they knew had worked in practice. None of it seems based on Danish research, but parts are based on North-American research. The chapter on reviews, for instance, is based on Michael Fagans work [Fagan, 1976 and 1986].

The SPU method is well known and often used in industry. The reason is that SPU gives a good framework for development without forcing you to give up the techniques you know from earlier projects. Most of the developers we interviewed admitted that they did not follow the SPU phase model strictly. The method was merely used as common basis for defining a project specific development method.

However, the SPU method does not offer much of a solution to many serious problems found in industry. For example: (1) SPU does not help enhancing traceability. (2) SPU does not help you define requirements, although it gives you checklists to evealuate a requirements specification once you have defined it. (3) SPU does not help you identifying parallel processes or deriving the task structure in a real-time system. With SPU, tasks are merely the highest level of module decomposition, thus linking it to functionality rather than concurrency.

In one large company they had implemented an extended SPU-model with the help of a consultant. They tried to follow the extended model strictly, but realized later that they now produced so many specifications and reports that they were quite sure nobody would ever read them. They asked a question we could not answer: How can we use a well-defined and rigid methodology for development without producing several meters of binders

with documentation every time we make a minor revision of a product?

The extended SPU method gave a very bureaucratic development where trivial things tended to obscure the more important issues. Real projects needed much more flexibility than the method allowed.

5. Other barriers

Education, new employees, supplier courses, seminars etc. are often mentioned as sources of knowledge and technology transfer. The interviews showed that these are very ineffective ways of transfer:

(1) Newly educated computer scientists may know a lot, but it is difficult for them to apply it on real projects, and it takes time for their knowledge to spread in the company where they get a job.

(2) Education is given a low priority by employees and management due to time pressures. Education only takes place on employee initiative. Formal initiatives, like career planning or education planning, are non-existent.

(4) Courses succeed when they deal with specific techniques that are immediately useful. But when one person from an organization attends a more general course, it does not work very well. This is because the person cannot use what has learned, and cannot talk to other employees about it. The newly gathered knowledge will just be left on the book-shelf upon return - no matter how good this knowledge may be. The intermediaries mention a typical cycle of 2 years for the introduction of a new technique. First, the companies send out a few technology-scouts, and if they decide to seriously adopt the technique, they either have a company-class set up, or they send off all the developers.

6. Suggestions for improved technology transfer

There are several players involved in the software technology transfer. Each could do something to improve the transfer, either alone or together with other players.

6.1. Researchers

The researchers could establish study groups which follow industrial projects, partly to learn more about practice and partly to get better inspiration for new relevant research.

The researchers could also try to link existing research more closely with specific problems in the electronic equipment industry. For instance, a lot of good Danish research in object orientation lack examples of protocol specifica-

tion, multi-tasking, process control etc. Industry will not consider using this research without such examples.

6.2. Industry

Companies must try to overcome the risk aversion if they are to perform better than competitors. At present they tend to believe that the ideal method will turn up some day. If this should really happen, they will see proofs, meaning that many other companies would have used the method for years before the method is taken into use in the Danish company.

A more realistic approach is to analyze the existing problems in depth and then take steps to remedy them one by one. At present, companies have a surprising lack of in-depth understanding of their own development problems.

Industry could also make more use of two transfer techniques which we have seen succeeding in other technical disciplines:

(a) Industrial Ph.D.-students paid half by the company and half by the government. The Ph.D.-student is expected to do research corresponding to a normal dissertation, and at the same solve a specific problem in the company. Industrial Ph.D.-students are used with considerable success in the development of electronic hardware, whereas we have only met very few examples within software technology.

(b) Bring in researchers to help solve specific problems. Some researchers in other technical areas are occasionaly contacted by companies who want advice within the researcher's area. Examples of this have been assistance in connection with control algorithms and assistance in estimating whether a design is satisfactory. Both parties express their satisfaction with this kind of requested assistance. However, industry claims that very few researchers in computer science are able to render this kind of assistance.

6.3. Intermediaries

Some intermediaries have a lot of success running so called ERFA-groups ("Experience Working Groups"). For example ElectronikCentralen is running a *Data Technical Forum* with, at present, 8 ERFA-groups. The participants in the groups are employees from the 50 member-companies representing all major electronic equipment companies in Denmark. The Forum may include lectures or company-visits. There is a widespread belief that the ERFA-groups are a good idea, allthough the results achieved in the groups vary a lot. Surprisingly, no researchers are members of the ERFA-groups. An obvious idea would be to offer researchers free or very inexpensive membership.

6.4. Education

The educational system today fails to teach state-of-the-art in software technology. For example, requirements specification, quality assurance, and performance calculation are not taught in sufficient depth to be practiced.

6.5. Government

The government could change the way researchers merit themselves. Today meritation is primarily linked to publishing scientific papers. Instead, government could make a point of rewarding industrial application of research results. The government could also support cooperation between industry and research such that researchers could apply their research ideas, and industry could reduce the risk of using such innovation. Finally, the government could encourage researchers to take a sabbatical year or two in industry.

Acknowledgements

We greatly appreciate the support from the Danish Scientific-Technical Research Council, which persuaded us to conduct the study. Richard Baskerville (from SUNY Binghamton) has provided many comments that helped us clarify the paper substantially.

References

Bak, Lars, et.al.: An overview of the Mjølner BETA system. In: F. Long (ed.): Software Engineering Environments, vol 3, Ellis Horwood Ltd, Chichester, England, 1991.

Biering-Sørensen, S. & F.O. Hansen & S. Klim & P.T. Madsen (1988): Håndbog i Struktureret Program-Udvikling. Teknisk forlag, 1988.

Elmstrøm, René (1989): A taximeter design project using CEDER. IFAD, Odense, November 1989.

Fagan. M.E. (1976): Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal, no. 3, 1976.

Fagan, M.E. (1986): Advances in Software Inspections. IEEE Transactions on Software Engineering, vol. SE-12, no. 7, july 1986.

Madsen, O. Lehrmann & B. Møller-Pedersen & Kristen Nygaard: Object-oriented programming in the BETA programming language. Addison

Wesley, June 1993.

Miles, M.B. & M. Huberman (1984): Qualitative Data Analysis: A sourcebook of new methods. Sage Publications, Newbury Park, California, 1984.

Rischel, H. & B. Graff Mortensen & A.P. Ravn (1987): Konstruktion af formålsbundne systemer. Teknisk Forlag, 1987.

Scrøder, B. & S. Lauesen & J. Pries-Heje (1993): Software til Apparater: Industri kontra Forskning. Department of Informatics and Management Accounting, Copenhagen Business School, ISSN 0903-6571 93/1.