Maintainability Myth Causes Performance Problems in Parallel Applications

Daniel Häggander, PerOlof Bengtsson, Jan Bosch, Lars Lundberg
University of Karlskrona/Ronneby
Department for Software Engineering and Computer Science
S-372 25 Ronneby, Sweden, Phone: +46 457 787 00, Fax: +46 457 271 25
[Daniel.Haggander | PerOlof.Bengtsson | Jan.Bosch | Lars.Lundberg] @ipd.hk-r.se

Abstract

A challenge in software design is to find solutions that balance and optimize the quality attributes of a system. It is not always possible to maximize each attribute and one has to make trade-offs. In this paper we present a case-study of an application where a key component, special designed to be highly maintainable, caused an unexpected and serious performance problem. Interviews with the developers show that the choice of design was based on a general assumption that a fine grained adaptable component design gives the system higher adaptability and thus also higher maintainability.

In this study we have implemented a prototype of an alternative design based on rigid but exchangeable components. This alternative component has been evaluated and compared with the original component with respect to performance and maintainability. The evaluation shows that the alternative component has much better performance characteristics as well as higher maintainability. These findings show that the original design decision was based on a general assumption that proved invalid, i.e. the performance problems in the application were caused by a myth.

Keywords: Performance, Maintainability, SMP, Object oriented design, Frameworks, Multithreading.

1. Introduction

Software quality requirements are getting more and more attention with the dawn of the software architecture discipline. It is not always possible to maximize each quality attribute in a design, and in that case one has to do trade-offs. Maintainability and performance often serve as examples of inherently conflicting quality requirements where a trade-off is inevitable [13]. In our experience, this is the general opinion in research as well as in the software development industry.

We have assessed a parallel commercial telecommunication system from Ericsson, a fraud control centre (FCC), for performance and maintainability independently to see if there was a conflict. The system is designed using object-oriented techniques and the parallel execution is implemented using threads [10]. The system seemed very suitable since the two driving requirements during its development were performance, i.e. throughput & scalability, and maintainability, i.e. adaptation of the design to new requirements. Our assessments show that the design of one component, a parser, was a key factor for performance as well as for maintainability.

Fine grained adaptable object designs (e.g. design patterns [4]) are generally considered to be more maintainable than designs based on rigid but exchangeable components. Interviews with the designers related to the project showed that this was also their opinion in the design of the FCC, particularly for the parser component. However, fine grained adaptable designs tend to be larger than designs based on rigid but exchangeable components. This was the main reason why, maintainability assessments of an alternative design showed that the assumption above does not apply universally and we therefore reject it as a myth.

Performance evaluations showed that the performance loss due to choosing the fine grained adaptable design instead of the design based on rigid but exchangeable components was very large, particularly when using multiprocessor platforms. Interviews with the developers show that these performance problems were not anticipated. Consequently, the design decision did not provide the anticipated leverage with respect to maintainability, and it caused an unexpected and serious performance problem. Thus, this case presents an excellent instance of a general misconception about maintainability and performance.

The rest of this paper is structured in the following way. The next section presents the FCC system. Section 3 describes our method, consisting of implementation and evaluation of an alternative design and interviews with the designers at Ericsson. In Section 4 we present the result from the evaluations of the alternative design and compare those results with the assessments of the original design. In Section 5 we discuss myths and reality regarding maintainability and performance. Section 6 concludes the paper.

2. Fraud Control Center (FCC)

2.1. Cellular Fraud

When operators first introduce cellular telephony into an area, their primary concern is to establish capacity, coverage and signing up customers. However, as their network matures financial issues become more important, e.g. lost revenues due to fraud.

The type of fraud varies from subscriber fraud to cloning fraud. Subscriber background and credit history check are the two main solutions to prevent subscriber fraud. In subscriber fraud the caller uses a false or stolen subscriber identification in order to make free calls or to be anonymous. There are a large number of different approaches to identify and stop cloning. FCC is a part of an anti-fraud system which combats cloning fraud with real-time analysis of network traffic [12].

2.2. System Overview

Figure 1 shows an overview of the total FCC system. Software in the switching network centers provides real-time surveillance of suspicious activities associated with a call. The idea is to identify potential fraud calls and have them terminated. However, one single indication is not enough for call termination.

The FCC application allows the cellular operator to decide certain criteria that have to be fulfilled before a call is terminated. The criteria that can be defined are the number of indications that has to be detected within a certain period of time before any action is taken. It is possible to define special handling of certain subscribers and indication types. An indication of fraud in the switching network is called an *event* in the rest of this paper.

The events are continuously stored in files in the cellular network. With certain time intervals or when the files contain a certain number of events these files are sent to the FCC.

In FCC, the events are stored and matched against predefined rules. If an event triggers a rule, a message is sent to the switching network and the call is terminated. It is possible to define an alternative action, e.g. to send a notification to an operator console.

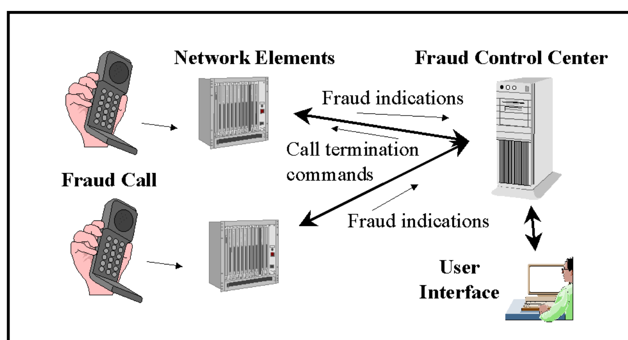

Figure 1. The FCC system.

The FCC application consists of five software modules, all executing on the same computer (see Figure 2). The TMOS module is an Ericsson propriety platform that handles the interaction with the switching network (2a), i.e. it is responsible for collecting event files and for sending messages about call terminations. The collected events are passed on to the next module (2b), which is the main module of FCC. In the Main module the files of events are parsed and divided into separate events. The complete module, including the parser, is designed using object-oriented techniques and design patterns. The events are then stored and checked against the pre-defined rules using module three, the database (2c). If an event triggers a rule, the action module is notified (2d). This module is responsible for executing the action associated with a rule, e.g. a call termination. Standard Unix scripts are used to send terminating messages to the switching network via the TMOS module (2e). The last module is the graphical user interfaces from which the FCC application is controlled (2f).

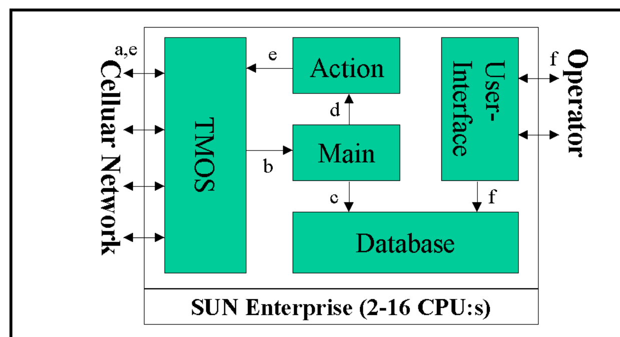

Figure 2. The five modules of FCC.

2.3. Performance Issues

The rapid growth in the telecommunication market increases the performance requirements on applications supporting telecommunication networks. In FCC, these performance requirements are meet by using a Symmetric Multiprocessors (SMP) [6], e.g. SUN multiprocessors. However, it is not trivial to increase the performance using SMP:s because bottlenecks can easily limit the performance significantly.

Performance evaluations showed that dynamic memory management becomes such a bottleneck in FCC [7]. The same evaluations showed that the parser was the limiting component. Dynamic memory management has in several evaluations of parallel applications been pointed out as a major bottleneck, especially for application implemented in C++ [3] [11] [8].

In C++ [15], dynamic memory is allocated using operator new which is an encapsulation of the c-library function malloc(). Allocated memory is deallocated using the c-library function free() via operator delete. Many implementations of malloc() and free() have very simple support for parallel entrance, e.g. using a mutex for the function code. Such implementations of dynamic memory result in a serialization bottleneck. Moreover, the system overhead generated by the contention for entrance can be substantial [8]. The bottleneck can be reduced using an optimized re-entrant implementation of malloc() and free(), e.g. ptmalloc [16]. However, even better performance can be achieved by decreasing the number of heap allocations [8]. A lower number of allocations will also improve the performance on

uni-processors. Using a re-entrant implementation of malloc() and free() will only increase SMP performance.

In an object-oriented design, especially with design patterns, a large number of objects are used. An object often requires more then one dynamic memory allocations (call to operator new) in its construction. The reason for this is that each object often consists of a number of aggregated or nested objects. For example, a car can be represented as a number of wheels objects, a car-engine object and a chassis object. A car-engine-object may use a string object, usually from a third-party library, for its name representation and so on (see Figure 3).

It is not rare that these objects are combined at run-time in order to be as adaptable as possible. For example, a car could have an unspecified number of wheels. Such a design requires that the each sub-object (Wheel) is created separately. As a result of this, dynamic memory are allocated and deallocated separately for each sub-object. The total number of allocations is, according to the discussion above, dependent on the composition of the object. Therefore every design decision which affects the composition of an object will also affect the number of memory allocations and deallocations during program execution.

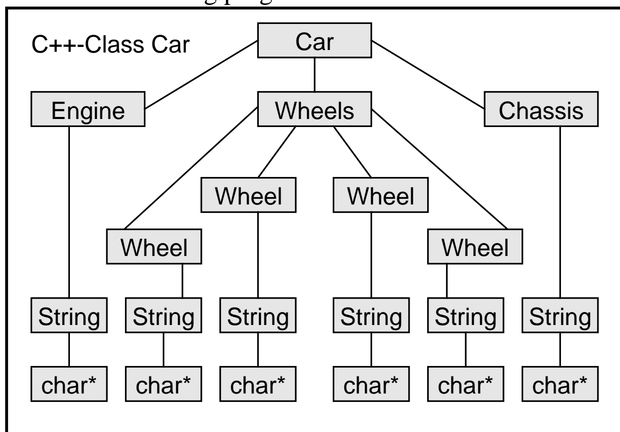

Figure 3. An object representation of car.

The parser component in FCC is similar to the car example above. The parser is constructed as a *State pattern* [4] where each possible state is represented by a class. A parser object holds one instance of every state class, in FCC's case nine. The nine instances are all created using operator new. All states also contain a third party software component; the "regular expression object", see Figure 4.

Consequently, the performance of the FCC application is seriously limited by the large number of dynamic memory allocations and deallocations generated by the design of the parser.

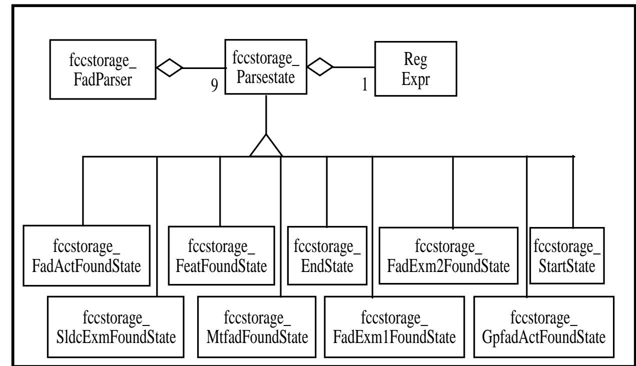

Figure 4. The object design of the FCC parser.

3. Methods

In Section 2.3 we described how congestion within the dynamic memory handling significantly reduces the performance of the FCC. We also suggested that a fine grained adaptable object design can be the reason for this.

Interviews with the people responsible for the FCC parser design, showed that maintainability requirements were the main reason for the fine grained design of the parser component. The interviews also showed that the designers were quite pleased with the design result, and they did not anticipate any substantial performance problems.

With the knowledge that fine grained designs can cause performance problems, we performed an assessment addressing the maintainability aspects of the FCC. Our intention was to find an alternative parser design which meets the maintainability requirements without causing performance problems. Based on the findings from this assessment, we suggested an alternative parser design, which we implemented and evaluated with respect to performance and maintainability.

3.1. Interviews

We performed interviews with four persons acquainted to the FCC in order to determine the reasons for selecting the fine grained design of the parser. Each person was interviewed separately from the others. We selected four persons using the following criteria:

1 The main designer for the product.

2 One of the persons implementing the parser

3 A person in the project not responsible for the design or implementation of the parser.

4 A person outside the project but from the same product area within the company.

This selection enables us to find out; what was the intended design decisions from the main designer's point of view; does the person implementing the component have the same point of view, does another programmer in the project have the same understanding of the decision even

when not directly involved and does this apply for other projects.

We performed the interviews using a questionnaire to help us remember all questions. The interview was divided into two parts. First part was questions to be answered in a more elaborate way. For example, one questions was "*Why did you choose to design the parser like you did?*". Second, a part with very direct questions to be answered only yes or no, very rapidly. For example, "*Did you choose the parser design to reduce implementation time?*".

3.2. Maintainability Assessment

We used a modified version of the scenario based software maintainability prediction method [2] on the software architecture of the FCC to assess the maintainability. In short the method consists of the following steps:

1 Identify categories of maintenance tasks

2 Synthesize scenarios

3 Assign each scenario a weight

4 Estimate the size of all elements.

5 Impact analysis

6 Calculate the predicted maintenance effort

The only difference to the method as presented in [2] was that we did use a ordinal scale *{Not applicable, Low, Medium, High}* for estimating the amount of code that (1) needed to be modified and (2) the amount of new code that needed to be written.

A person closely related to the FCC project synthesized fourteen (14) scenarios to a scenario profile. Below is an example scenario from the profile.

"Event format changed to include several fraud indications"

Size estimations and impact analysis were combined into one activity and graded on the ordinal scale (above). A score was calculated by calculating a sum of the impact on each scenario in the scenario profile. A lower score indicates better maintainability.

Besides the original design, a number of possible designs alternatives were evaluated using this scenario based maintainability assessment.

3.3. Alternative Parser Design

The maintainability assessment of the FCC software architecture resulted in a proposal for an alternative design of the parser component which seemed promising, both from a maintainability and a performance point of view.

Instead of designing an object oriented adaptable parser, an alternative is to design a special purpose parser for each input format. When using this approach, instead of having one parser component the system would include one special purpose parser and determine what parser to run for the particular input that arrives. A parser is a well known component in the compiler community [1] and several tools for generating parser implementations from grammars in Backus-Naur format (BNF) exists, e.g. the standard Unix tools Lex and Yacc.

The input format is suitable to describe as a context free grammar in BNF and should be easy to do. That would enable generation of several parsers for the time it takes to design, implement and test the original flexible parser. Not mentioning the reduced complexity of maintaining the system. A new format would require some few changes to the syntax specification, a few lines of code and the generation of a new parser.

A developer was given the task to implement an alternative parser, using the architecture described above. The input to the developer was, besides the architecture strategy, the requirement specification from the original FCC project. Furthermore, the developer was instructed to implement the parser quick and simple, e.g. without considering the possibility that the source code has to be adapted to new requirements in the future.

The development of the experimental parser would incorporate two main activities, first learning to use the Lex and Yacc tools and then the parser was designed and implemented. The developer had no prior professional experience with Lex or Yacc before the assignment. The prototype development did not include design documentation and only limited verification of the parser was made. These two aspects have to be taken into account when comparing the time effort for the alternative parser with the original FCC parser. The new implementation did not use multithreading. Instead, ordinary UNIX processes was spawned off in order to be able utilize all processors on SMP:s hardware. This was a direct result of using the Lex and Yacc tools, which can not be multithreaded.

3.4. Performance Comparison

The source code of the original parser was extracted from the FCC application and executed separately.

The performance of the two parser designs was evaluated on a SUN enterprise with eight processors, hosting a SUN Solaris operating system.

The throughput capacity was measured as the number of events parsed per second using one to eight processors. Since, the number of dynamic memory allocations has a large impact on the final FCC performance (see Section 2.3) the number of malloc() calls was counted for the two versions. The results of the measurements are presented in Section 4.

3.5. Maintainability Comparison

We compare the two different designs based on the following research results:

- Software maintenance effort have been shown empirically to be strongly correlated to the code volume [9].

- Productivity for software development of new code is higher than the productivity of modification of existing code. Productivity being (>6x) higher when writing or generating new code (>250 LOC/Man-month in C [14]), compared to modifying legacy code (1,7 LOC/Man-day ~ <40 LOC/Man-month [5]).

We make the comparison by measuring the code volume for the two parser implementations, i.e. the original FCC parser and the alternative parser we implemented. Then, since the code volume is strongly correlated to the effort, we calculate an effort prediction based on the productivity measures and the code volume measures.

If the code volume is significantly different in the two cases it is reasonable to assume that the comparison model will be valid. If the code volume measures of the two cases are very similar other factors might have to be involved in the comparison.

4. Results

4.1. Interviews

These are the results from the interviews. We have concatenated the answers to the questions and in the cases where the interviewed persons have disagreeing answers, we present both.

These are the answers regarding the original flexible parser:

1 *Why did you choose to design the parser like you did?*
Answer: We expected several input formats and late additions. It was a good way to introduce design patterns and it was a good object-oriented design.

2 *What alternatives were there?*

Lex & Yacc? Not familiar with the tools. Did not like to use generated code.

One hard-coded parser for every format? This was not adaptable enough. Some were personally against it.

A parser framework? Not considered.

3 *What consequences/advantages did you expect?*

Answer: More cost-effective, and less effort to implement support for new input formats even during the development project.

4 *What results have been achieved?*

Answer: The parser is fairly easy to adapt to new input formats.

5 *How would you solve the same problem now, based on your experiences now?*

Answer: Basically the same way, using the state-pattern, but the input formats have changed even more than anticipated and more flexibility and adaptability would be good.

The results from the direct questions in the interviews are presented in Table 1. P1 through P4 is the persons interviewed, where P1 is the main designer for FCC, P2 is the parser programmer, P3 is another project programmer, and P4 is a designer from another project within the same product domain.

Question	P1	P2	P3	P4
Did you choose the parser design to reduce implementation time?	N	N	N	N
Did you choose the parser design to enable easy adaptation of the FCC to new or changed requirements?	Y	Y	Y	Y
Did you choose the parser design to improve the performance of the FCC?	N	N	N	N
Did you choose the parser design to minimize error sources?	Y	N	N	N
Did you anticipate any performance bottlenecks at all from these designs?	N	N	N	N

Table 1: The results from the interviews

The result from the interviews show that the fine grained design was selected for maintainability reasons, and that no performance problems were anticipated. The answer to question 5 shows that the designers had themselves not been able to connect the performance problems in FCC with the parser design, i.e. the connection is obviously non-trivial.

4.2. Measurements on the Original Parser

The original implementation of the FCC parser has 3889 LOC (C++) and makes 100 calls to malloc() for each event file parsed. The maximum throughput is, using one processor 1800 events per second and using eight processors 680 (see Figure 5 and table 2). This gives a speed-up less than one. We have observed this kind of behavior before in previous projects, and the reason for the poor speed-up is congestions within the dynamic memory handling [8]. According to Ericsson the time effort to develop the original parser was 350 hours.

4.3. Measurements on the Alternative Parser

The experimental implementation was developed using the standard Unix tools, LEX and Yacc. The number of LOC (Lex, Yacc and C++) was 433. The parser is based on a nine (1+8) UNIX processes architecture. The number of malloc() calls is eleven for each file of events parsed. The maximum throughput is, using one processor 6800 events per second and when using eight processors 55600 events per second (see Figure 5), i.e. somewhat more than eight times speed-up. The work load generator interferes more when only a few processors are used, and results in this su-

per-scalar speed-up. However, this effect is marginal and the speed-up is almost linear.

The time effort to develop the alternative parser was approximately 2 days. A compression with the original of the FCC parser can be seen in Table 2.

Consequently, the alternative parser has much better performance, particularly when using multiprocessors. Moreover, the development time for the alternative parser was much shorter than for the original, and the code size was also much smaller for the alternative parser.

Parser	Arch.	LOC	# CPU	Max events/sec.	# malloc calls
FCC	multi-thread	3889	1	1800	100*event + 84
			8	680	
Alt.	UNIX-proc.	433	1	6800	11*event + 19
			8	55600	

Table 2: Experiment results

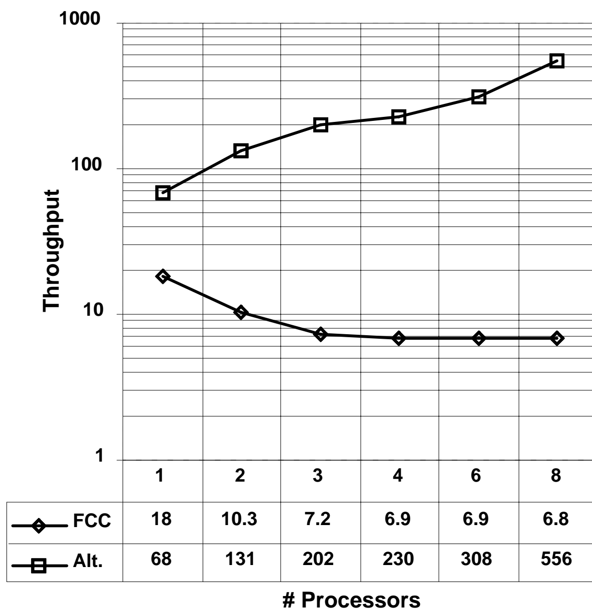

Figure 5. The throughput for the original and the alternative parser.

5. Discussion

Today, object oriented design and design patterns have become best practise in industry. Many designers are trained in state-of-the-art object oriented design and very familiar with design patterns. The object oriented design research community conveys that flexibility and adaptability are best achieved by designing components using design patterns. It is being taught and appreciated as a generally applicable assumption that guide the design of modern industrial applications.

Associated with this common assumption, it is assumed and accepted that a flexible and adaptable solution will have lower performance than a rigid exchangeable component design, but one expects that the performance reduction will be limited or even marginal.

In the Ericsson FCC project, the parser was identified as a key component for maintainability as well as for performance. In their efforts to produce a design that enabled them to adapt the parser component to new requirements the designers introduced a major performance bottleneck in their system. Interviews with the designers clearly show that the parser design was chosen to increase the adaptability, i.e. the designers adopted the general assumption that making a component adaptable would increase the overall system maintainability. By focusing on a single component in the system they overlooked an alternative. By making the system adaptable to changes and designing the component to be exchangeable new requirements could be met by replacing the component with a new one. The use of design patterns resulted in a component that is clearly adaptive. However, it requires nine classes and ~4 kLOC.

The results from the measurements of an alternative parser design show that the source code for that parser was about a tenth of the size of the original parser implementation. In [5] it is shown that the maintainability of a software module is highly correlated to its size in lines of code. Since the alternative design of the parser component was implemented in significantly less lines of code than the adaptable parser, we believe the maintainability effort for our alternative to be significantly less. Measurements of the implementation time for the original parser (350 h) and the alternative parser (16 h) strongly support this.

Hence, the assumption that an adaptable component design will increase the maintainability of the system proved to be invalid in this case. Therefore, the general assumption that fine grained adaptable designs are more maintainable than designs based on rigid but exchangeable components does not apply in all cases, and we reject it as a myth.

Performance comparisons between the original and the alternative design show that the alternative design has a much higher performance, particularly when using a multiprocessor. The throughput using eight processors is more than 80 times higher for the alternative design when using a multiprocessor with eight processors, and we expect that the performance difference will be even larger when using more than eight processors. The interviews show that the designers agreed unanimously that they had not anticipated the performance bottlenecks in their original design. The designers had themselves not been able to connect the performance problems in FCC with the parser design, i.e. the connection is obviously non-trivial.

Performance measurements of the FCC system showed that the parser component was the limiting bottleneck in the original design [7]. The original parser design resulted in a large number of object creations needed for parsing, thus increasing the of use of dynamic memory significantly. Con-

sequently, the general assumption above, which we rejected as a myth, affected the performance of the FCC in a negative way. Hence, the performance problems in FCC were caused by a myth.

In this study we have found no characteristics or other evidence that distinguish this project from the general conception of other industrial development projects.

6. Conclusions

It is a challenge to find solutions that balance and optimize the quality attributes of a software system. It is not always possible to maximize each attribute and one has to make trade-offs. In this paper we have addressed the importance of the design decisions made during software development and the consequences of basing those decisions on assumptions or even prejudice.

We present a case, based on a real telecommunication application, where the performance has been significantly limited because of the assumptions that an fine grained adaptable object designs (e.g. design patterns [4]) would give the application higher maintainability.

An alternative design was implemented and evaluated. The evaluation results show that the alternative parser has much better performance characteristics as well as higher maintainability. These findings show that the design decision was based on a general assumption that proved invalid, i.e. the performance problems in FCC were caused by a myth.

The contribution of this papers is that the current practise in object oriented design and design patterns is not always the best alternative for neither performance nor maintainability. A design based on rigid but exchangeable components can give better performance as well as higher maintainability.

Acknowledgments

We would like to express gratitude to the persons in the FCC project who participated in the interviews and to Ericsson Software Technology AB for giving us the opportunity to study their work.

References

[1] A. Aho, R. Sethi, A. V. Aho, J. D. Ullman, Compilers, *"Principles, Techniques And Tools"*, Addison Wesley Longman Inc., 1985, ISBN: 0201100886

[2] P. Bengtsson, J. Bosch, *"Architecture Level Prediction of Software Maintenance"*, in Proceedings of 3rd European Conference on Maintenance and Reengineering, Amsterdam 1999.

[3] R. Ford, D. Snelling and A. Dickinson, *"Dynamic Memory Control in a Parallel Implementation of an Operational Weather Forecast Model"*, in Proceedings of the 7:th SIAM Conference on parallel processing for scientific computing, 1995.

[4] E. Gamma, R. Helm, R. Johnson, J. Vilssides, *"Design Patterns"*, Addison-Wesley, 1997.

[5] Henry, J. E., Cain, J. P., *"A Quantitative Comparison of Perfective and Corrective Software Maintenance"*, Journal of Software Maintenance: Research and Practice, John Wiley & Sons, Vol 9, pp. 281-297, 1997

[6] K. Hwang and Z. Xu, *"Scalable Parallel Computing"*, McGraw-Hill, 1998.

[7] D.Häggander and L. Lundberg, *"Multiprocessor Performance Evaluation of a Telecommunication Fraud Detection Application"*, submitted, 1999.

[8] D.Häggander and L. Lundberg, *"Optimizing Dynamic Memory Management in a Multithreaded Application Executing on a Multiprocessor"*, in Proceedings of the ICPP 98, 27th International Conference on Parallel Processing, August, Minneapolis 1998.

[9] W. Li, S. Henry, *"Object-Oriented Metrics that Predict Maintainability"*, Elsevier Publishiing, New York, Journal of Systems and Software, v23, n2 , November, 1993, pp. 111-122.

[10] B. Lewis, *"Threads Primer"*, Prentice Hall, 1996.

[11] L. Lundberg and D. Häggander, *"Multiprocessor Performance Evaluation of Billing Gateway Systems for Telecommunication Applications"*, in Proceedings of the ISCA 9th International Conference in Industry and Engineering, December, Orlando 1996.

[12] Catharina Lundin, Binh Nguyen and Ben Ewart, *"Fraud management and prevention in Ericsson's AMPS/D-AMPS system"*, Ericsson Review No. 4, 1996.

[13] J.A. McCall, Quality Factors, *"Software Engineering Encyclopedia"*, Vol 2, J.J. Marciniak ed., Wiley, 1994, pp. 958 - 971

[14] K. D. Maxwell, L. Van Wassenhove, S. Dutta, *"Software Development Productivity of European Space, Military, and Industrial Applications"*, IEEE Transactions on Software Engineering, Vol. 22, No. 10: OCTOBER 1996, pp. 706-718

[15] B. Stroustrup, *"The C++ Programming Language"*, Addison-Wesley, 1986.

[16] http://www.cs.colorado.edu/~zorn/Malloc.html