Mälardalen University Licentiate Thesis
No.38

# ARCHITECTING SOFTWARE FOR COMPLEX EMBEDDED SYSTEMS - QUALITY ATTRIBUTE BASED APPROACH TO OPENNESS

**Goran Mustapić**

**2004**


Department of Computer Science and Engineering
Mälardalen University

ISSN number: 1651-9256
ISBN number: 91-88834-74-3
Printed by Arkitektkopia, Västerås, Sweden
Distribution: Mälardalen University Press

# ABSTRACT

There has been such an increase in the complexity of systems embedded in modern industrial robots, construction equipment, cars, trains, systems in telecommunication networks etc. that the engineering of such systems requires the coordinated efforts of tens or hundreds of engineers. These engineering efforts are multidisciplinary, requiring software engineers, control engineers, hardware engineers etc. to work together in the design and implementation of the systems concerned. Many have a lifetime of 10-20 years which means that their maintenance must be carefully planned. Furthermore, most of the systems can be classified as safety-critical or dependable systems. The importance of software in complex embedded systems is increasing and software quality issues are becoming first level concerns. One of the keys to success in industry in the future will be the ability to develop high-quality embedded systems and their software on time in order to remain competitive. The complexity and size of these systems requires a systematic approach to software architecture analysis and design. To advance research in this area, we need to improve our understanding of software architecting for these systems. Among the disadvantages of current methods of software architecture design are that they target “pure” software systems and that they make simplifying assumptions that software architecture design of embedded systems begins from a well-defined, fixed list of requirements for a software subsystem.

This thesis is divided into two parts and has two main objectives. In the first part, we describe the state of practice of software architecting of complex embedded systems. The objective here is to get a better understanding of the factors that influence the software architecture design of complex embedded systems. In the second part of the thesis, we study industrial robots, as open, complex embedded systems that can be extended and programmed by third party. The objective is to determine how software quality concerns and in particular dependability concerns can be systematically approached when system openness is increased.

The main contributions of the thesis are the following. Several important common factors which influence the architectural design of complex embedded systems software are identified, the state of the practice is described and some areas that require further research are identified. In considering the design of open robotics system, the thesis shows which dependability means need to be applied in the architecture design phase, to support system openness. The dependability means are systematically applied within the context of software architecture approach based on quality attributes.

The results presented in this thesis are based on industrial experience from the author’s work as an engineer-researcher and from systematically performed interviews with several industry experts having positions as senior architects or similar in companies that develop complex and long-lived embedded systems.

# ACKNOWLEDGMENTS

I want to thank to my academic supervisors Ivica Crnković, Christer Norström and my industrial supervisor Peter Eriksson very much. They were always available for countless number of discussions on the journey towards this thesis.

Thanks to all of my colleagues at Mälardalen University, Department of Computer Science and Engineering and ABB Automation Technologies AB, Robotics and especially to those who co-authored and reviewed the papers that were the basis for this thesis. In particular, I want to mention Anders Wall and thank him for his detailed reviewing. Thanks to the managers at ABB Automation Technologies, who supported my work, especially to Lennart Sundstedt and Kent Vasberg. Additionally, I want to thank Johan Schubert for the discussions about the terminology and methodology used in this thesis.

Thanks to my parents, my brother and sister for their support and encouragement. And, to save the best for the last, thank you Carina, Filip and Klara for being with me every day.

Goran Mustapić, Västerås, November 2004

# TABLE OF CONTENTS

# 1 Introduction

*"My third piece of advice (to researchers) is probably the hardest to take. It is to forgive yourself for wasting time. … As you will never be sure which are the right problems to work on, most of the time that you spend in the laboratory or at your desk will be wasted. If you want to be creative, then you will have to get used to spending most of your time not being creative, to being becalmed on the ocean of scientific knowledge."*

*Steven Weinberg*

## 1.1 Background and Motivation – the Industrial Context

The work on which this thesis is based was performed in conjunction with real world industrial projects at ABB Automation Technologies AB, Robotics division.

Industrial Robots, like many other industrial systems, are complex business-critical systems demanding exceptional operational qualities such as reliability, availability and safety. For instance, industrial robots are required to have a MTBF (Mean Time Between Failures) of 50000 hours. Some robot models are powerful machines that can handle loads up to 500 kg and can possibly cause substantial damage to their operating environment in the event of malfunction.

Advances in computer hardware contribute to the increasing power of embedded systems. They place new design and implementation demands on software and more and more advanced features are now being realised in software. For example, robots can pick up an object, move it in different directions and determine many of its physical properties; robots can detect the collision of a robot arm with an object in its environment; vehicles have advanced controls that reduce the need to over-dimension physical parts, vehicles have safety features such as ABS brakes and diagnostic interfaces for external tools. These are just some examples of software taking increasingly important roles in systems. The software required is becoming increasingly complex and there is an increasing need for systematically dealing with software as the valuable asset. Software quality issues are becoming first level concerns in these systems as a key to success in the future will be the ability to develop high-quality systems and their software on time [20].

The presence of external equipment and their long lifetimes make realistic experimenting with these systems difficult. We must instead rely on a good understanding of the systems and on methods for predicting their properties. The nature of these systems is such that multidisciplinary approaches are often required to the development of a system. Control experts, process experts and software engineering experts are often required to work together on

system design and implementation. Software architecture design is the first step in the process of developing software, a high-level solution being designed to satisfy many different and often contradictory requirements. For this thesis, our goal was to get a good understanding of the most important factors that influence software architects when they make design decisions. There is a lack of such studies in general and especially for complex embedded systems, where software is embedded in a system which is the primary concern, not the software itself. The first goal of this thesis is to obtain a clear picture of the context and current practices of software architecting in a complex embedded system environment.

Industrial Robots are by their nature generic systems which can be used in many different industrial environments and perform many different tasks. There is a wide variety of mechanical units that perform these different tasks. The heart of the robotics system is its control system hardware and its software. The ABB robotics software system is built on the key principles of product line architecture and is the key enabler for making the robotics systems highly customizable, extensible and programmable by third parties. For example, the systems have a clearly recognizable platform, which is one of the key assets of product line architecture. We characterize these systems as "open systems" and within the scope of this thesis the terms *open* and *openness* are defined more precisely in the following way:

> *An open system is a system that has extensibility mechanisms that make it possible to independently, across organizational and company boundaries, develop and choose components of the system, resulting in new system functionality. Openness is a property of open systems.* [1]

As distinct from an open system, a closed system is a system with no public extensibility and programmability mechanisms for use by third parties. Why is this kind of openness interesting in the first place? One of the main reasons is that it is a key enabler for innovation and for building solutions that are close to end-customer needs, by reusing key core common assets.

ABB have the improvement of the functional and non-functional properties of their Robotics system as a permanent objective. One goal is to make the system *more open*. When its openness is increased, more possibilities are given to third parties to program, customize, and extend the system. At the same time, the quality of the system and company responsibility to end customers remains an important concern. Enabling different levels of system openness requires different means and techniques for providing support. For example, Microsoft Windows as an open platform has very different techniques for supporting new device drivers as opposed to supporting scripting languages that run in Internet Explorer. Previous research on

[1] In other contexts, the term *open system* is often used to refer to systems that support "open" standards and that are open in the sense of interoperability, standards compliance and the possibility of integration with other systems. Integration is often performed with the purpose of creating a larger system. This interoperability based definition of open systems should not be confused with that used in this thesis.

non-functional requirements and software quality teaches us that proper support for quality must come when a change is introduced, not as an afterthought.

During a system lifetime, the system and its environment must evolve to reflect changing conditions and new requirements. Introducing openness and changing the level of openness of a system is one type of change that is of interest in this thesis. It is possible to transform system architectures on the basis of previous experience of architects or one of the systematic approaches available can be selected and used in the process. However, finding an appropriate method to guide systematically the design process is difficult as many different approaches exist and the experience of using the methods in different domains is still relatively limited. Many available software development techniques do not take into account the specific needs of embedded-systems development [20]. The main goal of the thesis is to investigate which dependability means[2] need to be applied to support system quality when openness in the robotics system is increased and if these dependability techniques can be systematically applied using a software architecture approach based on quality attributes.

## 1.2 Research Questions

The purpose of the research in this thesis is to contribute to the solving of the real-world problem, not to solve the real-world problem; there is a clear distinction between the goals of the research project and the goals of a real-world industrial project. There are many different aspects of the real world problem, many questions to be answered, many of which are not research questions. On the contrary, the solution of a practical problem can be important input to the validation of the research project's products and ideas.

A good understanding of the industrial context is needed in order to avoid oversimplification of the problem, neglecting certain important factors and being unaware of important influences and relationships. Therefore, the first main research question addressed in the thesis is directed towards *characterization and generalization* of software architecting context in complex embedded systems:

> ***Q1:*** *What factors are considered important by software architects of complex embedded systems and have significant impact on software architecture design?*

*Factors* are any facts that are likely to constrain or otherwise influence the architecture [52]. Analyzing the wider context of complex embedded systems, rather than analyzing industrial robotics systems only, gives us an opportunity to study similar systems. We may thereby recognize important factors otherwise neglected, missing, or discounted when analyzing a

[2] Means for dependability are "the methods and techniques giving the system the ability to deliver a service conforming to the accomplishment of its function and to place a trust in this ability [30]." (for an overview of Dependability and its terminology see Appendix A2)

single system or type of systems. Analyzing the role of a software architect in these systems is related to this question and is also a subject of investigation.

The second main research question is targeted towards investigating the *applicability of methods or means of development* in providing assistance in the context of the industrial project at ABB. The question is stated in the following way:

> ***Q2**: Which dependability means should be applied on software architecture level to support system quality in an open system and what is a suitable approach to systematically applying those dependability means?*

A suitable method should provide assistance in understanding and dealing with tradeoffs between increasing openness in a system and other important factors and in particular product quality. Furthermore, the approach should include a systematic process that guides the transformation of architectural design.

## 1.3 Outline of the Thesis Contributions

The thesis and its contributions to the store of software engineering knowledge are structured in two parts. This reflects the fact that there are two main research questions. Only the main contributions of the thesis are listed in this section. A more detailed discussion together with answers to the research questions are included in the conclusions of the thesis (Chapter 5).

The main contributions of the first part of the thesis are related to software architecture design for complex embedded systems:

- Systematic capture of the state-of-practice of software architecture design for complex embedded systems. In complex embedded systems, components of the system architecture are likely to be subsystems consisting of hardware and software. The hardware may be custom made and hardware costs are likely to put constraints on the resources available for software. Hardware-software version compatibility should be carefully planned as both hardware and software are subject to version changes, etc. The work we have done is based on data provided by the architects of several different systems in the following domains: telecommunication, robotics, and transportation and construction equipment.
- Several heuristics of good and bad practices in architectural design are identified, such as the importance of the core architecture team, its deliveries of and relationships with architectural work and individual projects.
- Certain areas in which industry still has no generally accepted systematic approaches and could benefit from future research are identified. One such issue is communication of architectural decisions and principles both during initial design and evolution.

The second part of the thesis looks into various aspects of openness and the applicability of quality attribute based software architecture design methods in the context of complex embedded systems. The contributions are summarized as follows:

- The thesis contributes to a general understanding and definition of non-functional system property designated *openness*.
- There is no generally accepted and unique way of modelling quality concerns in system analysis and design. Separation of system qualities to design-time and operational qualities is often suggested in literature [12,49]. It is shown in the thesis, how this separation is used in the design of an open system.
- The thesis provides an example of how quality attributes based software architecture evaluation in combination with dependability means is used in the design of an open system that fulfils quality concerns. We identify a potential for improvement of the methods in better integration between system and software level.

## 1.4 Research Methodology

The overall research design, upon which this thesis is based follows the conceptual research design framework described by Robson in [51] and guidelines by Shaw [53] [54]. The most important elements of the research design are depicted in Figure 1.

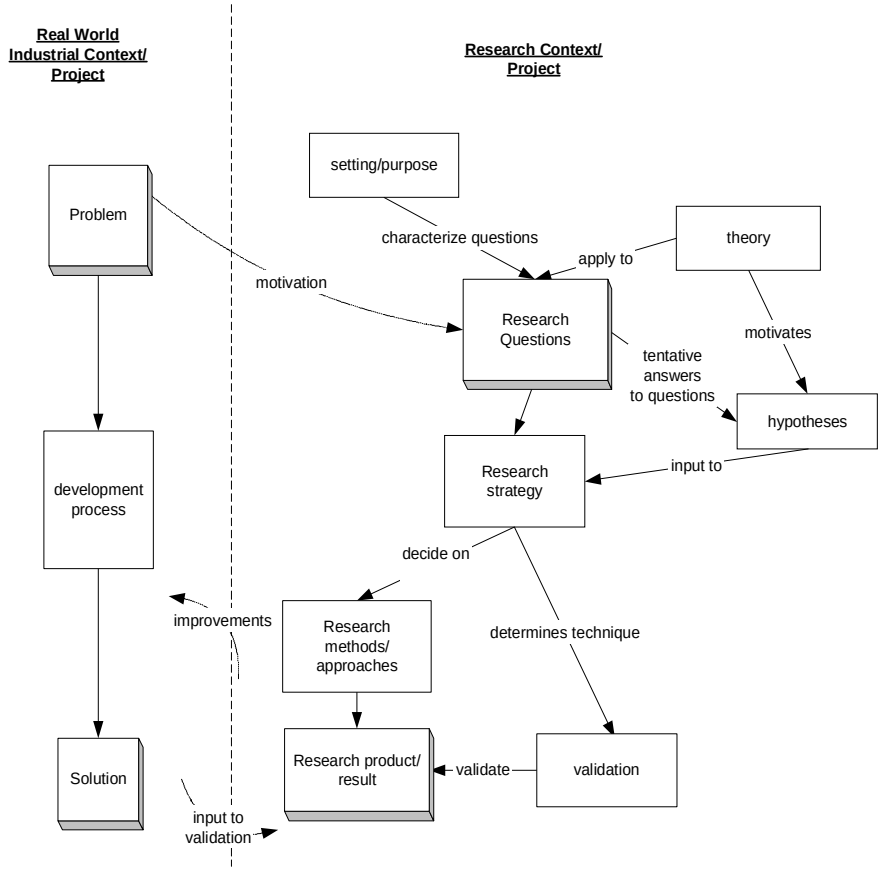


**Figure 1 Elements of design of a "real world" research project (based on [51])**

The starting element of the research context is research *questions* that are motivated by the real-world problem. Questions *settings* determine the type of the questions. The most common settings are: feasibility, characterization, method/means, generalization and selection [53]. In this context, the *theory* is a basis for predicting or choosing a particular answer – the *hypothesis*. Whether a hypothesis is formulated at all and whether it is expressed more or less strictly/formally, depends on the type of questions and the theory. *Strategy* is the overall strategy of the research project. Determining the strategy involves: choosing between *fixed (quantitative)* and *flexible (qualitative)* research strategies (such as case studies or grounded theory), deciding on the type of research method to be used, how results will be validated etc. In a fixed research strategy, most of the details of the research method are determined in advance. However, in certain cases, especially when the research goals are related to a project whose

goals are evaluation and change, it is difficult to make all the detailed decisions regarding the research strategy in advance. Flexible research strategies are more suitable for those kinds of situations. In a flexible research design, all of the elements in Figure 1 are revisited and refined during the enquiry. The detailed design framework emerges during the enquiry. *Methods* are particular tactics for collecting the data (e.g. interviews), data analysis and analysis of the trustworthiness of the results.

### 1.4.1 Theory and Research Hypotheses

There is an enormous number of possible answers to our research questions (Section 1.2). The search for possible answers and solutions must be limited in some way so that a meaningful enquiry can be performed. Choosing the theory (a part of the existing knowledge in the area of software engineering), which is used to formulate the hypothesis (predicted answers) to the questions is one of the crucial points in the design of the research project. Possible risks are e.g. not searching enough for a suitable existing theory or choosing an inadequate theory. Parnas says that one of the problems with research in software engineering and computer science is that "principles may be harder for people to apply than expected" and that "there is no glory in confirming other people's results; you have to invent your own" [44]. In the case of this thesis, the uniqueness of the industrial context makes it very hard to find an exact match of a theory that is proven to fit the problem description and the given context.

The nature of the problem and the research questions asked, are primarily from the research area of *software architecture and dependability*. In the scope of this thesis, *dependability means* and *quality attribute based software architecture design methods* are used as "theories". There is also a strong relationship between the research questions and the areas of: software quality, requirements analysis and modelling, and of course to the general context of systems and software engineering. Figure 2 shows the most important research areas related to the research questions. Even though it is not illustrated in this figure, there are certain overlaps between the individual research areas.

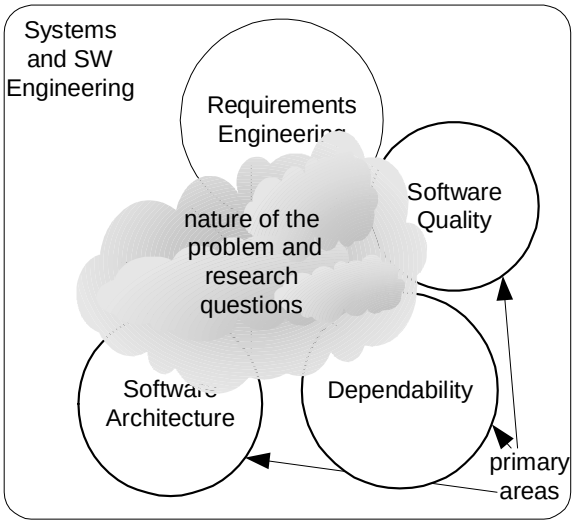


**Figure 2 Research areas of concern for the problem**[3]

In general terms, using a research hypothesis is most applicable in fixed design research. In a fixed design we are in a good position to predict answers before the actual data are gathered, while in flexible research we are more likely to be able to make these predictions afterwards. Therefore, a simple definition of hypothesis as a predicted answer to the stated questions is suggested in [51]. As it will be described in the next section, we have chosen the flexible research strategy.

Having this discussion in mind, we formulate the following hypotheses:

> ***H1****: In addition to the general factors that are considered in the architectural design of software systems, the factors that originate from relations between software, hardware and entire system must be taken into consideration.*

> ***H2****: Quality attribute-oriented software architecture design methods can be used as a systematic approach in applying dependability means in the context of open business-critical complex embedded systems.*

Note that we have chosen to formulate a hypothesis for the Q1 and a hypothesis for the second part of Q2 question (i.e. "*…what is a suitable method …*"). No hypothesis is formulated for the

[3] Overviews and the most important terminology and concepts from these research areas which are relevant for questions discussed in this thesis, can be found in Appendix A.

first part of Q2 (i.e. "*Which dependability means…*"), but we have rather chosen to provide an answer to this question directly based on the research results.

### 1.4.2 Research Strategy, Methods and Results Validation

The main research strategy we have chosen is a flexible or qualitative research. There are two main reasons for choosing the flexible research design – the nature of the questions and the author's position as engineer-researcher. The nature of the questions is such that they belong to the domain of software architecture and software engineering in which people are necessarily involved in the context of investigation. There are many parameters that cannot be influenced or controlled during the enquiry. The hypotheses must be evaluated with respect to a realistic problem in a real world context. Action research is very well suited, particularly for Q2 and H2, for case studies in which the purpose of enquiry and evaluation may result in a change of existing practices. The author's position as engineer-researcher fits well in this context. Working part time in industry and part time as researcher, provides an ideal background for participative case studies or action research.

***Research method for question Q1 and hypothesis H1:*** A case study was performed to collect sufficient information to be able to support hypothesis H1 and give a more detailed answer to the question Q1. Data collection in the case study was based on: interviews, mail-exchanges and a workshop with embedded system experts. In order to give a structure to the discussions, we proposed several discussion topics, such as the influences of the following factors on software architecture design: relationships between system, computer hardware and software architecture design, reuse/inheritance/legacy influences, business/application domain, etc.

The research result is a qualitative or a descriptive model that is based on the factors that influence software architecture design. The validity of the research results and conclusions is addressed in the following way. First, mail exchanges were performed to clarify uncertainties from the interviews. Second, the interview results were critically reviewed and analyzed by the research group and third, the interview results were discussed in a workshop with the interviewees. Finally, the results were presented at a major software architecture conference and reviewed by a group of international experts.

***Research method for question Q2 and hypothesis H2***: *Quality attribute based methods* from *software architecture* and *dependability means* were applied in realistic projects, in which openness was an important consideration. This part of the research project is based on active participation as an engineer-researcher in two industrial projects at ABB Robotics. The research method used is participative case study and the data is collected through discussions with team members and observation of practices.

The research results in this case are well-organized observations and an analysis of the applicability of quality attributes-based architecture design principles to software architecture design in a complex embedded system. The validity of the results for Q2 is less certain than that of the results for Q1. Being too close to development projects, in the role of an engineer-

researcher, may result in the drawing of incorrect conclusions and biased results. A strategy for producing valid results includes: real-world experience from discussions with industrial project colleagues (informal interviews), parallel discussions with academic supervisors and reviewing and publishing results in the international research community. Critical thinking is essential for producing valid results.

## 1.5 List of Publications

A list of publications that form the basis for this thesis is presented below. As the publications are results of the team work of researchers, my main individual contributions are highlighted.

**Articles in collection**

1.) *A Dependable Open Platform for Industrial Robotics - A Case Study*
Authors: **Goran Mustapić**, Johan Andersson, Christer Norström, Anders Wall, published in the book:
*Architecting Dependable Systems II*
Editors Rogerio de Lemos, Alexander Romanovsky and Cristina Gacek, published by Springer-Verlag, Oct 2004.
Individual contribution: the main author. The case study is based on the industrial project Open Controller in which I and partly Johan participated as engineers-researchers.

**Conferences and workshops**

2.) *Real World Influences on Software Architecture - Interviews with Industrial Experts*
IEEE Working Conferance on Software Architectures Oslo, June 2004. IEEE
Authors: **Goran Mustapić**, Anders Wall, Christer Norström, Ivica Crnković, Kristian Sandström, Joakim Fröberg, Johan Andersson
Individual contribution: the main author. The paper presents an analysis of the data from the technical report 5) below.

3.) *A Dependable Real-Time Platform for Industrial Robotics*
In ICSE 2003, WADS, Portland, OR USA , May 2003.
Authors: **Goran Mustapić**, Johan Andersson, Christer Norström
Individual contribution: the main author. A position paper which I presented at the WADS workshop to get research peer feedback regarding the problem statement.

4.) *Propagation of quality attributes in a layered design*
SERPS'03 Third Conference on Software Engineering Research and Practice in Sweden, Lund, October 2003
Authors: **Goran Mustapić**, Ivica Crnković
Individual contribution: the main author. A case study based on an industrial project (Teach Pendant device for industrial robots).

**MRTC reports**

5.) Influences between Software Architecture and its Environment in Industrial Systems – a Case Study
MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-164/2004-1-SE , Mälardalen Real-Time Research Centre, Mälardalen University, February 2004
Authors: **Goran Mustapić**, Anders Wall, Christer Norström, Ivica Crnković, Kristian Sandström, Joakim Fröberg, Johan Andersson
Individual contribution: main author and editor, participated in all interviews but but one, mail-dialogs with interviewed architects to clarify uncertain issues, presentation of interview analysis results at workshop.

6.) Component Based Software Engineering for Embedded Systems - A literature survey
MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-102/2003-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, June 2003
Author(s): Mikael Nolin, Johan Fredriksson, Jerker Hammarberg, Joel G Huselius, John Håkansson, Annika Karlsson, Ola Larses, Markus Lindgren, **Goran Mustapić**, Anders Möller, Thomas Nolte, Jonas Norberg, Dag Nyström, Aleksandra Tesanovic, Mikael Åkerholm
Individual contribution: survey of three papers.

7.) Modern technologies for modelling and development of process information systems
MRTC Report ISSN 1404-3041 ISRN MDH-MRTC-100/2003-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, May 2003
Author(s): Ivica Crnkovic, **Goran Mustapić**, Mikael Åkerholm
Individual contribution: minor contribution.

The thesis includes publications 1) and 2) with changes of style type (numbering, references etc). The thesis also includes a modified version of publication 4).

## 1.6 Thesis outline

The outline of this thesis follows the flow of the research method described in Section 1.4.2. In addition to this Introduction and Appendix A) containing a description of the research areas and the terminology used, the thesis is divided into three main parts. In Part I we consider the real life context of software architecting in complex embedded systems. Part I is based on publication 2) [39]. In Part II, research results based on involvement in industrial projects are presented. Publication 1) [38] is included in Chapter 3. Publication 4) [37] is basis of Chapter 4 and it has been modified based on the feedback at the conference and experiences gained after the conference. The thesis completes by Part III that contains conclusions, discussions of results validity, and a proposal for future work.

# PART I - UNDERSTANDING IMPORTANT FACTORS THAT DETERMINE SOFTWARE ARCHITECTURE OF COMPLEX EMBEDDED SYSTEMS

*Not everything that can be counted counts and not everything that counts can be counted.*

*Albert Einstein*

The main goal of this part of the thesis is to present the research results that are used to answer and support question Q1 and hypothesis H1. Chapter 2 is based on the published article [39]. The published article has been extended to include the questionnaire table used at interviews (Section 2.2), previously found in the technical report ([40]). Comments from other architects we have met at the conference are included in Section 2.6.

In the published paper we use terminology “industrial systems”, but in this thesis we have decided to use more generally accepted terminology “complex embedded systems.” The terminology that is used in the paper that is included in Chapter 2 is preserved in the original published form.

# 2 Real World Influences on Software Architecture – Interviews with Industrial Systems Experts


## Abstract

Industrial systems are examples of complex and often long-lived systems in which software is playing an increasingly important role. Their architectures play a crucial role in maintaining the properties of such systems during their entire life cycle. In this paper, we present the results of a case study based on a series of interviews and a workshop with key personnel from research and development groups of successful international companies in their Swedish locations. The main goal of the investigation was to find the significant factors which influence system and software architectures and to find similarities and differences between the architecture-determining decisions and the architectures of these systems. The role of the architect was an important subject of the investigation. Our findings result in recommendations relating to the design and evolution of system architectures and suggestions regarding areas in which future research would be beneficial.


## 2.1 Introduction

There are many large and complex software-intensive systems which have been successful, not only in commercial terms, but in providing reliable bases for several products over many years. One of the key factors in successfully managing a system, i.e. maintaining the system, introducing new features etc. is its architecture. There are several factors that distinguish the management of large and complex industrial systems from the management of smaller systems. The requirements of smaller systems can often originate with and be understood by a single person, while the requirements of large systems have many dimensions and involve many stakeholders, which makes them much more complicated and difficult to grasp and manage.

It is not feasible to study the important architectural factors affecting large systems by constructing and reasoning about small “toy-systems”. The architectural work in real systems has so many different aspects, that it is unrealistic to experiment on system models and expect to draw conclusions of value. Large and complex systems can have a lifetime of 20-30 years, which makes experimenting with these systems even more difficult. A realistic study of factors important for successful system management requires that we can study a system over a long period of time, or at least have access to a reliable source of information regarding the history of the system.

We have studied seven large and complex industrial systems in which software has an important and expanding role. We looked at several aspects of these systems related to their architecture at different points in their lifecycles, i.e. requirements, design and implementation

of the system, system evolution and retirement. Among many others, the following issues are analyzed in their relationship to architectural decisions: reuse, legacy, the effects of the choice of technology, standards, organization and development process.

Even though we have primarily focused on software architecture, it is important to have in mind that a typical industrial system incorporates computer hardware, other hardware and software. One of our goals was to find out how much importance is given to software architecture in the design of these systems. In [7], the authors "persist in speaking about software architecture primarily, not system architecture, …, because most of the freedom is in software choices, not hardware choices." However for the systems we have studied, the following statement by Maier and Rechtin [32] may be more appropriate: "Even if 90% of the system-specific engineering effort is put into software, … it is the system, not the software inside that the client wishes to acquire."

The systems we have studied are: the electronic control systems of cars and construction equipment provided by Volvo Car Cooperation and Volvo Construction Equipment respectively, a robot control system provided by ABB Automation Technologies AB/Robotics, a train control system provided by Bombardier Transportation, the software system of radio base stations for 3G provided by TietoEnator Telecom & Media, and the Ericsson telecom system and platforms of some of its nodes. The study is based on interviews with specialists in the system and software architectures of the companies' system and software development organizations. The results from the interviews were summarized and further discussed during a workshop in which the interviewed architects participated.

Most software architecture-related research is focused on architectural analysis, architectural descriptions, and tools. Consequently, most of the documented case studies are focused on these issues [55] [13]. Other reports present the utilization of related architectural activities in the software development process in terms of descriptions and analysis [57]. A case study similar to ours but of relatively limited scope is presented in [20]. We have found no other relevant work that addresses important factors related to the software architecture in successful software intensive industrial companies to the same extent as this paper.

The contribution of this paper is a description of the state-of-practice in industry with respect to software architecture and a set of observations and findings from an analysis of interviews and a workshop with the chief architects from these companies. The findings include architecture-related differences between and similarities of the systems studied. Moreover, we provide speculative findings and trends that were not explicitly found in the case study, but were rather the results of the intuitive reasoning of the interviewers.

The outline of the paper is the following: in Section 2.2 we present a detailed description of the method used in performing the interviews. In Section 2.3 we give a brief overview of the systems studied. Section 2.4 presents a comparative analysis of the data collected in our case study. Section 2.5 contains our conclusions and some suggestions for our future work in this field. Some feedback to the published results is included in Section 2.6. We express our gratitude to the individuals who participated in the project and their companies in Section 2.7.

## 2.2 The case study setup

The case study performed was an investigation of the architecture of software. According to [45] the following steps are involved in a case study: conception, hypothesis setting (particularly important as it describes what we measure and how the results are analyzed), design, preparation, execution, analysis, dissemination and decision-making. Also, because a case study usually compares one situation with another, some measures must be taken to avoid bias and to make sure that hypothetical relationships are tested. Possible techniques are: sister project, comparison with a general baseline and random selection. We have performed these general steps in the following way. In the preparation phase of the case study we held several brainstorming sessions. The outcome of these meetings was a plan with the following agenda:

- Formulate a list of question to guide the interview.
- Request and arrange interviews with people who have an architect, chief designer or similar important role and know the history of the system. The interview was estimated to take at least two hours, and was conducted by one or preferably several of the authors of this report. We decided to use the following criteria in the selection of companies to participate in the case study:
    - Successful products on the market
    - Complex, established industrial products in which software is an important part of the entire system
    - Availability of specialist architects
    - Availability of data from several system generations
- After the interviews, write summaries to send to the architects concerned for review.
- Assemble the results of the interviews in a technical report.
- Make a preliminary analysis of the interviews and determine the similarities of and the differences between the cases.
- Organize a workshop with the interviewees and perform a further analysis of the results.
- Publish the results of the case study as a scientific article.

The purpose of submitting a questionnaire was to give a common structure to all the interviews and to form a basis for comparison and analysis of the results. The layout of the questionnaire was the following:

**Table 1Questionnaire for architecture interviews**

1. Relationship of software and system architecture and propagation of requirements between them;
    a. How would you order influences between:
        i. Software
        ii. System
        iii. Hardware

        architectures (e.g. which one is determined first, second and third)?
    b. How are influences between SW and system architecture communicated (e.g. through requirements, what notation is used etc)?
2. Reuse/inheritance/legacy issues;
    a. Code
    b. Subsystems
    c. Experiences
3. Business/application domain factors:
    a. Standards
    b. Requirements
    c. Type of customer
    d. Volumes
    e. Length of life
    f. Non-functional requirements (NFR)
4. Choice of technologies:
    a. Who influences who and how (architecture and technologies)?
    b. In what extent?
    c. How to handle a mix of technologies?
    d. How do possible future technology changes influence architecture?
5. Organizational issues and architecture;
    a. Who influences who and how?
    b. Work distribution and allocation.
    c. Distributed development.
    d. Are there third party contractors? If so, how does it influence the architecture?
    e. How to balance project and competence organization?
    f. Size (5 or 5000 developers needed to implement the system)?
    g. Dynamic (how often does the organization change).
    h. Maturity of the development organization and people competence
6. Process issues;
    a. What type of development process do you use and why?
    b. Does process influence architecture and how?

7. Resources in architectural activities
    a. Calendar time for architectural activity
    b. People involved in architecture related activity
8. Other

The interviews were not restricted to the questions in the questionnaire. Wherever possible, we advanced counter arguments to provoke discussion, well aware that some of the statements were a summary of the interviewee's experience with multiple systems over a period longer than 10-15 years. In some cases, we have conducted several interview sessions to penetrate different subjects and e.mail exchanges to clarify some uncertainties.

## 2.3 Systems overviews

In this section, we present a brief overview of the systems covered by the case study. The systems have been categorized in the following groups in accordance with their engineering domains: industrial automation-robotics, telecommunications, construction and transportation. The way we present the system overviews reflects the views and responsibilities of the interviewees.

### 2.3.1 ABB Automation Technologies/ Robotics

ABB Automation Technologies Products – Robotics is a manufacturer of industrial robotic systems. The interviewee has been the chief system architect for the two most recent system generations. Industrial robots are systems consisting of one or more mechanical units (robot arms that can carry different tools), electrical motors, robot controller (computer hardware and software) and clients (used for on-line and off-line programming of the robot controller). Industrial robots can be characterized as generic tools that can be configured and programmed for a specific purpose such as painting, welding, palletizing etc.

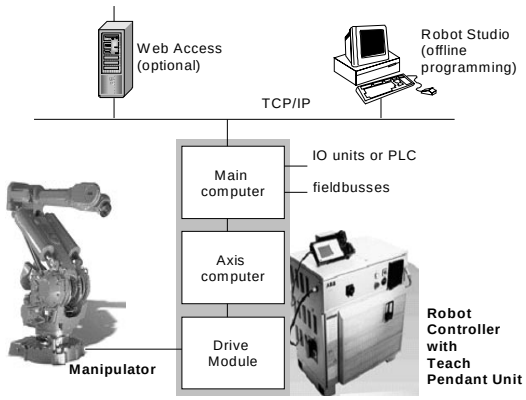


**Figure 3 System hardware view of an industrial robot system from ABB Robotics**

A hardware view of a robotics system is shown in Figure 3. The clients of the robot controller are optional, while the main computer and axis computer are tightly coupled, with real-time constraints on their communication. The main computer executes user programs and generates the path to be followed (the coordinates) and then sends these to the axis computer, which controls the axes of the manipulator.

The system has evolved through four generations, and the fifth generation of the system is currently being developed. Compared with the first generation (S1), which used the first microcomputer-based electrical robot control system, and whose software design required about three man months of work, the effort required to develop the software for the fifth generation (S5) is estimated to be about 100 man years. In the most recent system-generation shift, even though the computer hardware of the robot controller was completely changed, the software architecture in the system is considered to be the same, since the basic software infrastructure and patterns have not changed, even though much new functionality has been added.

One of the initial requirements was that the same controller should be used for all the different types of robots, and thus the architecture can be characterized as product line architecture. More than 60K S4 generation controllers are currently in use.

In essence, the controller has layered architecture and within layers, an object-oriented design. The implementation consists of approximately 2500 KLOC of C language source code divided into 400-500 “classes” and organized in 8 technical domains. The software platform of the robot controller defines the infrastructure that provides basic services such as: a broker for message-based inter-task communication, configuration support, persistent storage handling, system startup and shutdown, etc. These basic services constraint the implementation of the software system, as defined by the architecture.

### 2.3.2 Ericsson AB (R&D System Management)

The interviewee is a member of the R&D System Management group, the task of which is to maintain an overall technical view of how the Ericsson product portfolio and platform technology is evolving. The subject of our discussion was the Ericsson telecom system as a system of systems. Telecom systems have a long history and have been standardized to permit a global communication system. Standards are the dominating factor for system functionality in this domain - a sign of the maturity of the telecom industry. Packet switching networks are more recent and these systems are still subject to more dynamics and changes in their requirements. A telecom system is a complex system and an example of only a part of a telecom network is shown in Figure 4. The figure shows a radio access network, the radio base station acting as the radio modem, converting digital information to analog radio signals and vice versa. This particular part of a telecom network was illustrated because the software architecture of one of its nodes is discussed in the following section.

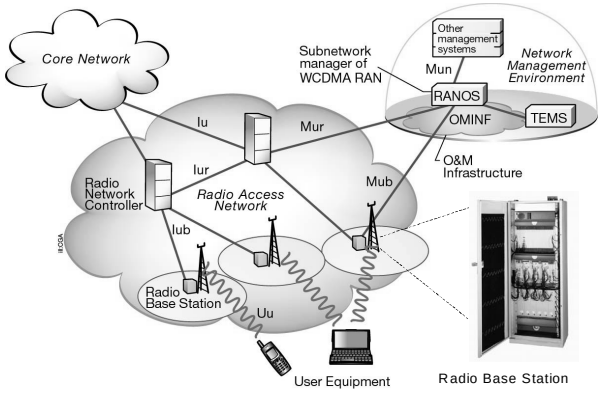


**Figure 4 Radio Access Network**

A telecom system requires exceptional reliability and availability. For example, a switch is started only once, and after that it should be possible to perform almost all maintenance during operations. Interoperability is another key property of a telecom system.

### 2.3.3 TietoEnator Telecom & Media

TietoEnator Telecom & Media, in partnership with Ericsson, develops 3G-base stations for the new mobile telecommunication system UMTS. We interviewed a senior software specialist who acts as one of the software architects of the Radio Base Station (see Figure 4). One of the most significant characteristics of 3G base stations is that they are sold in very large numbers

(for example, each 3G provider in Sweden requires about 12,000 base stations). They must have very high degrees of availability and ease of maintenance.

The system currently being developed is the first version of the new 3G-base stations. The development of the base station system was preceded by an initial prototype and experimental implementation, gathering experience and evaluating architectural solutions. In the actual product development, experience from the prototype development was very useful, but no software from the prototype was reused. The radio base station controller software consists of approximately 2000 KLOC of code organised in about 5000 UML-RT model elements, i.e. capsules, protocols and classes. About 80% of the code is generated automatically from Real-time UML. The base stations are built on a platform delivered by Ericsson (CPP, Connectivity Packet Platform). The platform is based on the OSE-Delta real-time operating system.

Figure 5 shows the functional architecture and the major functional components of a Radio Base Station. The functional components can be realized both in software and hardware, the Traffic Control and Operation & Maintenance being the most SW-centric components.

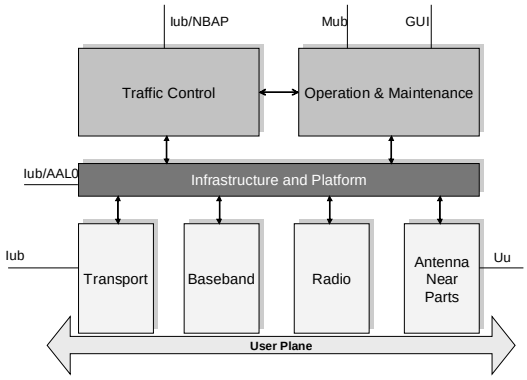


**Figure 5 Radio Base Station Functional Architecture**

The main characteristic of the software architecture is that the functional architecture has been arranged in a layered structure. The layering can be seen in three distinct dimensions: From the Traffic Control point of view, from the Operation & Maintenance point of view and from the Platform point of view. The main focus in the layered structure from the Traffic Control point of view is on the hardware abstraction layer, which decouples the higher layers from the actual HW realization in the Transport, Baseband, Radio and Antenna-near parts. The decoupling is achieved by means of reusable components that provide the User Plane functionality in the same way, irrespective of how the HW is being realized, i.e. irrespective of which kind of radio base station is being built.

### 2.3.4 Ericsson AB (Core Network Development)

In this interview, we met two software technology specialists at the Ericsson Core unit, Core Network Development. This group develops platforms which provide the basic hardware and software in different types of telecommunication systems. Examples of platforms are CPP, AXE switch platform and WPP platform (for GSN nodes). Platforms include both hardware and software. Examples of systems built on these platforms are nodes in telecom networks: Digital switching systems (e.g. AXE108), Mobile Base Stations (e.g. RBS discussed in Section 2.3.3 above), SGSN (GPRS Support Node).

Many architecture patterns can be recognized in each of those platforms, e.g. “client server", "blackboard" and especially “pipes and filters”. The system has a layered structure both on the system and subsystem levels. Special attention was devoted in the architectural design of these systems to concurrency and availability.

### 2.3.5 Volvo Construction Equipment

Volvo Construction Equipment (Volvo CE) develops and manufactures a wide variety of construction equipment vehicles such as articulated haulers, excavators, graders, backhoe loaders, and wheel loaders. We interviewed a technical specialist in electronic systems, who has been involved in the architectural design of several generations of the Volvo CE system.

Compared with passenger cars, most construction equipment vehicles are equipped with less complex electronic systems and networks. The focus in product development is somewhat different. The products are to be used at construction sites, and the most important requirement of the vehicle is to be a reliable machine to increase production.

Electronic control systems are important parts of the construction equipment product and are crucial for providing end-user functionality, such as automatic gearbox, engine and differential lock control, in addition to providing diagnostic and service functions.

Using a distributed electronic system reduces the cost of the product by permitting the use of sensors and displays for several purposes and enabling the use of control solutions which permit the use of les expensive mechanical components.

Figure 6 shows the basic architecture of the electronic system consisting of several ECUs (Electronic Control Units) connected by busses. A unified hardware is currently used for all nodes except for the display ECUs that differ due to space and appearance requirements. A unified hardware means, in this case, a common design with configurable I/O.

Together with the common hardware platform, Volvo CE uses a common software platform for the on-board ECUs. All of the nodes have a layered structure as shown in Figure 6. Software implementation that is reused between nodes includes: boot code, drivers, communication software, service software, error handling etc. Tools such as compiler, code generators, and scheduler can be used more easily due to the fixed hardware platform. Methods for parameterization of software are also reused.

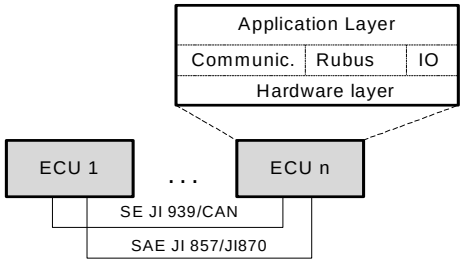


**Figure 6 Components in electronic system of Volvo CE equipment.**

To permit the reuse of software components and methodology in different products, Volvo CE has incorporated the Rubus component model for the real time application domain. The component model is an important part of the Volvo CE electronic platform since it enables reuse and commonality in terms of tools and methods. The Rubus component model is similar to the pipes and filters model.

### 2.3.6 Volvo Car Corporation

Volvo Car Corporation (Volvo CC) is a subsidiary of the Ford Motor Company, manufacturing a premium product aimed at the upper end of the car market. In this interview we met with the Program Manager, Research & Advanced Engineering. Volvo CC manufactures nearly half a million cars per year. To achieve these volumes, and still offer the customer a wide range of choices, the products are built on platforms containing common technology that has the flexibility to be adaptable to different models. As an example, the Volvo XC90, which appeared in 2002, is based on the same platform as four previous Volvo models launched since 1998. This reduces the development cost and makes it possible to reuse the same manufacturing facilities and strengthens the brand image through an increased similarity between the models.

A typical configuration of a Volvo CC car includes ECUs from more than 10 suppliers. A Volvo CC car contains a maximum of about 40 ECUs , connected via 4 different networks. In order to increase control in integrating supplier software and hardware components, Volvo CC uses methods and tools to assist in this effort.

The component technology is to a large extent provided by external suppliers, who work with a number of different car companies (or OEMs, original equipment manufacturers), providing them with similar parts. The role of the OEM is to provide external suppliers with specifications so that the component supplied will be suitable for a particular car model. Currently, external suppliers offer components in the form of ECUs with associated software, but as the computational power of the ECUs increase, it will be more common to include software from several suppliers in the same nodes, this increasing the complexity of the integration. The suppliers develop the ECU software using their tools and structure, but Volvo CC, as an OEM specifies: communication, power consumption, diagnostics, and software download procedure.

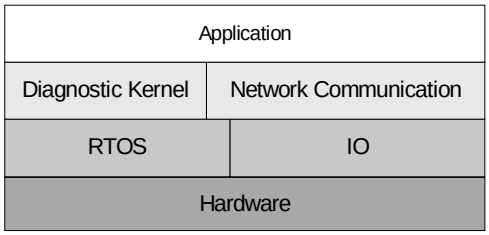


**Figure 7 ECU node architecture.**

The Volvo CC node architecture is a layered architecture (Figure 7) that must include a diagnostic kernel and a network interface provided by Volvo CC. All these components are integrated by Volvo CC, which is responsible for guaranteeing each node's resource requirements with respect to communication bandwidth.

### 2.3.7 Bombardier Transportation

Bombardier Transportation is the global leader in the rail equipment, manufacturing and servicing industry. We interviewed persons with two different roles – a system architect and a technology specialist. Examples of Bombardier products are: passenger rail vehicles and total transit systems, locomotives, freight cars, propulsion & controls, signaling equipment and systems. The group, representatives of which we met, develops components of the control system, propulsion and control systems. The control system components are delivered to the system groups that develop complete solutions for the end customers, and in particular, sub-domains (e.g. InterCity trains, Metro, Railway Control System).

A simple model of a train with the most important elements of its control system is shown in Figure 8. A standard designated TCN (Train Communication Network), defines the network interconnections between vehicles (WTB – Wire Train Bus) and within vehicles (MVB - Multifunction Vehicle Bus). A common time-triggered protocol is used on both WTB and MVB bus.

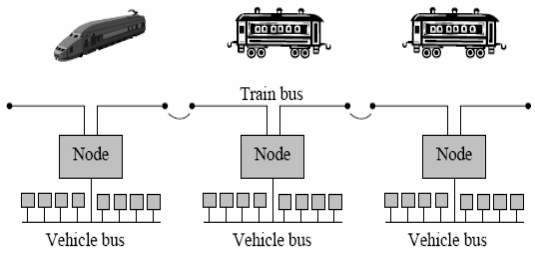


**Figure 8 Train communication network**

The layers pattern is dominant on the architectural level. Other design patterns such as publisher-subscriber and cyclic execution are also used in addition to own internal patterns to develop applications in a distributed deterministic real-time environment. For application programming, within nodes of the control system, Bombardier uses the IEC 61131-3 standard programming languages. The standard defines five languages, and of these *function blocks* (FCB) is that used mostly.

In addition to the traditional control system-related functionality, comfort-related functionality is becoming more important. Each of the computer-based electronic equipment units developed by the organization can be classified in one of the three domains of criticality: comfort (e.g. connection system, passenger information, entertainment, etc), safety (control system that is a part of the safety control) and control.

## 2.4 Comparative interview analysis

In this section, we present a comparative analysis of the interview data grouped according to the questions used in the interviews and listed in the Table 1.

### 2.4.1 Relationship of system, computer hardware and software architecture

From the brief overviews of system and software architectures presented in Section 2.3, we can conclude that they have many similarities. All of the systems have complex distributed system architectures with distributed and relatively autonomous units. In some cases we can treat them as systems of systems. The software architectures of system nodes are also similar (e.g. layering approach).

In [32] Maier and Rechtin discuss a shift in architectural design methods from “hardware first” to “software first”. One of the goals of our investigation was to determine the current state of practice regarding the relationship of software and system architectures in industrial systems. That the “software first” approach is becoming increasingly common is demonstrated by several of the systems we have studied; recent system generation has changed from “hardware first” to “software first” at both ABB Robotics and Volvo CE. Increasingly more intelligence is encapsulated in the software of these systems. In the ABB Robotics example, the computer hardware was completely redesigned in the most recent system generation shift, the software platform remaining the same. In the Ericsson cases, hardware still dominates the design of the software system. In the Volvo CC case, the hardware architecture remains dominant as it is the basis for integration of external functionality but the importance of software and software architecture is increasing.

The following are views regarding software vs. system knowledge expressed by interviewees. One stated that a “good enough knowledge, or at least understanding of the system and the HW... with respect to managing, supervising and controlling all the different HW is needed. However, I find that my experience in SW engineering and good general

knowledge of SW is much more valuable in getting a good SW architecture." Another commented "I have seen 'strange' software solutions built by application/domain specialists who did not have sufficiently good general software architecture design knowledge." This indicates that system expertise is necessary, but not sufficient.

### 2.4.2 Reuse and legacy in architectural design

As described in Section 2.3, several of the systems described have passed through clearly defined generations and within the systems, there have been corresponding generations of the (control) system hardware and software. We analyzed the relative importance of the following three factors: experience, subsystems and code, in the design of a major new generation of a system.

All of the interviewees stated that *experience* in developing similar systems, or previous generations of the system, was of the greatest importance. This is because the design of a new system generation seldom, if ever, begins with a blank sheet.

The reuse of subsystems in a new system design was considered to be an important economy and therefore can have more impact on the architectural design than might be expected. There are examples (ABB, Ericsson), in which complete subsystems were either reused from a previous system generation or acquired from a third party, when a new generation was designed. In the case of ABB, the design of the S4 system generation would have never been approved, because of the unacceptably high cost, if the new architecture required all subsystems to be changed or replaced. It is hard to disregard the legacy in long-lived systems. In the case of Bombardier, we have seen an example in which new hardware was designed with the explicit requirement that it should run old system software without change. In safety critical systems, e.g. train safety control, it is highly desirable to reuse a critical code that has been proven to work well in practice.

As the amount of software in a system increases it becomes increasingly important to reuse software components to minimize the investment in a new system or generation. It is therefore important to package software components in such a way that they can be used in a variety of architectures. A stable base platform infrastructure and easily reconfigurable connections between components on higher-levels is economically advantageous as seen in the ABB Robotics example. The software architecture in the system is considered to be the same in the most recent system generations, because although some new, very different features have been added, the basic patterns, message-based communication and similar have not changed. From a structural point of view, as connectors and components, this architecture would be considered to be a new architecture because a number of components and the way they are connected have changed. Of course different system properties, such as timing properties or reliability can be changed, due to changes in the structure, and must therefore be re-verified.

### 2.4.3 Business and application domain factors

In this section, we analyze the impact of business and domain related factors on system and software architecture. More specifically, we have investigated the influence of the following factors: standards, type of customers, production volumes, product lifetime, and non-functional requirements.

The influence of *Standards* varies on the basis of the domains in which the systems are used. Standards completely dominate the telecom domain, leaving space for competition based on optional features in standards and other non-functional requirements. Standards are not only used for interoperation between different systems, but also for interoperation between the nodes within a system. This is the case not only in telecommunication but also in the automotive industry in which e.g. the standardization of bus protocols is used as the integration point. The importance of standards and interoperability go hand-in-hand in the telecom domain. It is very important, to remain competitive in the market, to participate in standardization activities, to be at the leading edge and deliver solutions as soon as standards are finalized. For other systems and domains, we were told that standards have the greatest impact on subsystem level (e.g. a safety subsystem). This says basically that system partitioning was applied in the solution space, in order to comply with standards more easily (i.e. to certify a subsystem rather than the whole system). It is common to use software to implement more advanced safety features and have software independent core safety function as a backup (e.g. robotics system safety, ABS systems for car brakes, trains safety). Industry standards have a considerable impact on system architecture, e.g. for robotic systems - support for different kinds of field busses, for train control systems - TCN standard for train networks, other industry standards for network communication. These systems often have architectural level variability points to be able to support, for example, multiple industry standard protocols.

The *type of customers* has an appreciable impact on the process of architectural design. In some cases (Volvo CC) there are many small customers. In this case, the development partners are those involved in the architectural design. In other cases (ABB Robotics, Ericsson) there are both very large customers and many small. We were told that: “large customers have their opinions not only about what a system should do but also how a system should be built”. In the case of special customers, architectural issues may be a matter for discussion between the customer and the architects.

*Product volumes* also have an important effect on the architectural design. If a product is to be manufactured in large numbers, it is easier to justify the expenditure of more time in optimizing the product with respect to certain properties if this will lower the unit cost. The properties which need to be further optimized depend on the product. Ease of maintenance and fault tracing by non-experts is definitely one important property (Volvo CC, Volvo CE).

*Product lifetime* is another factor that is important in the architectural design of the systems we have studied. Most of the systems studied use a layered approach to decouple hardware and software, but also to decouple platform software and operating system. When the product

lifetime is 20-30 years, any unavailability of the OS used to build the original system is as much a problem as the unavailability of the hardware used in the original system design. We were shown examples of projects in which it was decided to use the Linux OS platform instead of MS Windows, because it was believed to be preferable to own the source code. We are not aware of any systematic approach to dealing with this issue.

We have seen that *non-functional requirements (NFR)* have a significant explicit impact on the architectural design of systems investigated. Each of the systems has particular non-functional requirements that are dominating and explicitly taken into account. Operational NFR are always analyzed from the system level. For some systems (e.g. trains) there is more focus on hardware NFR because of the nature of the operational environment (e.g. temperature variations –40C to +50C, humidity, vibrations, etc), while for other systems (e.g. telephone switches) software and hardware play an equally important role in providing support for performance/concurrency. The implementation of software concurrency mechanisms and fault-tolerance (especially fault isolation) is given much attention during the architectural design in telecom networks. Understandability of the system architecture was stressed by all of the interviewees as being very important.

### 2.4.4 Choice of technologies

A number of questions were asked during the interviews to determine how technologies and architecture have influenced each other in the creation process and the evolution of systems. In particular, we wanted to know if any explicit architectural activities or measures taken were related to choice of technology. We used the term technology in a broad sense - something that includes particular principles, methods and tools - e.g. database-technology, .NET or Java technology or similar.

From the architectural point of view, a common opinion was that technology does not play a crucial role. An ambition is to keep architecture separated from implementation, and in many cases, the use of a particular technology is seen as a matter of particular implementation. New technology is introduced carefully, very often in a part of a system (ABB Robotics, Ericsson). The introduction of new technology is sometimes forced by cost reduction requirements, e.g. the introduction of new cheaper hardware (Bombardier), or because of a possibility of utilizing more efficient tools (Bombardier, ABB Robotics).

However we have seen that the choice of technology may have important impacts on the architectural documentation, design and evaluation. If a particular technology brings some important advantages in analysis and settings of quality attributes (such as timing properties), its use becomes the central paradigm of the development process. One characteristic example is the use of a component-based technology used by Volvo CE in which the software and, to a degree, the system architecture are expressed consequently in terms of components with a standardized specification. Another example is the Model Based approach which is used at TietoEnator in the design of RBS software (Section 2.3.3). UML-RT specifications are strictly used and the code is

generated from the specification. In these cases technology plays an important role, achieving not only greater efficiency but also a better understanding of the system architecture.

A somewhat surprising finding related to technology was that technology choices are sometimes even made to motivate the developers of the system and create enthusiasm in the team.

### 2.4.5 Organizational factors and architecture

We analyzed the following issues in this section: influence of distributed development on architecture, outsourcing, size, maturity, dynamics, etc., of the organization which was to implement the system.

It is widely accepted that the organization influences the architecture; for example, Conway's Law says that the structure of the organization that builds some software matches the structure of the software [15]. The allocation of resources and people working on the project will have a direct impact on the architecture of the system. The majority of the interviewees stressed that the company's organization often mirrors the system architecture and vice versa. Proper handling of this relationship is important in order to minimize the dependencies in the software that make integration and validation more difficult and also to achieve distinct interfaces between different organizational units.

Several interviewees emphasized the importance of taking into account in architectural design, the fact that implementation will take place in geographically different places. We have also seen examples of distributed development not being taken into account, this resulting in less than optimal architectural support for the distributed development process. However, partitioning that is appropriate in a distributed organization may impair other system properties.

Several interviewees mentioned that changes in the organization are more frequent than changes in the architecture. Some of the reasons for organizational changes are company mergers and changes in the market. Moreover, some employees leave and others join the development organization and it is important that the newly employed people can become productive quickly. That is why NFR mentioned in 4.3 - *understandability* is of the utmost importance.

Some of the interviewees had fundamentally different opinions concerning understandability. An Ericsson architect opinion is that it is impossible to achieve quality if the developers do not "have the big-picture", i.e. understand how a sub-system is used and operates with other units. On the contrary, at ABB Robotics, opinion is that it is more important that many developers can be good development contributors, without knowing and understanding the entire system. This is because the ABB Robotics system is relatively multi-disciplinary; control engineers, mechanical engineers, and software engineers must be able to contribute to the same system without necessarily having any deeper knowledge in the other's fields. Other possible factors that have an impact on this issue are, for instance, related to the turnover of engineers, and the magnitude of the system.

One organizational and process issue related to system evolution which is particularly interesting is how to balance long-term strategic goals and short-term project goals. The conflict begins as early as in the architectural phase (experiences from Ericsson) – should more time be spent in establishing a good solid base for a long-term product evolution, or in getting a single product to the market as soon as possible?

### 2.4.6 Process related factors

There was surprisingly little interest among the architects (with a few exceptions) in life cycle and development processes. While it is obvious that architecture is a main means of preserving system properties in an evolution process, it seems that the process itself is not a direct concern of the architects. This implies that the mutual relations and influences are not under direct control but happen more or less ad-hoc. Examples in which a process view of the architecture would be beneficial are the designing of a testable architecture or, for managing future changes in, e.g. technologies, by having a life-cycle process associated with the architecture. We believe that the absence of a process view in these architectures may be one of their weaknesses and also that the indirect impact of the processes is appreciable. An example in which a more defined process could be advantageous was given by the architect at TíetoEnator. Integrating software and hardware for the very first time is usually a source of friction. One potential remedy of this problem could be software/hardware co-design.

Standards, e.g. safety standard such as IEC 61508 and ISO 15998, specify the development and maintenance processes. These standards will affect Volvo CE and have already been considered at Bombardier Transportation in different domains of criticality (comfort, control, and safety). Processes for development in the safety domain are much more strict and regulated, than e.g. those for the comfort domain.

### 2.4.7 Resources used for architectural design

Factors that were discussed included the time and effort invested in architectural activities and the number of people involved in architectural design.

The companies that participated in this investigation spend several years in developing a new architecture. Typically, architectural evaluation is based on prototypes and pilot systems. All the architects who participated in this investigation agreed upon the importance of small core-teams that decide the architecture, i.e. *single-mind-consistency*. Typically, architects should be people with knowledge of the domain, the technology, and the architecture. One interviewee described an architect as “a person who could implement the complete system by himself/herself, if only time (to market) permitted.”

The fundamental principles of the system i.e. its basic infrastructure are a result of the architectural design. Examples of fundamental principles in this context are the infrastructures for communication and concurrency. The fundamental principles should be developed and stable before too many developers are involved. We were given a contra-example of an

unsuccessful project (Ericsson), one of the main reasons identified being that too many people were involved before the basic principles were stabilized.

## 2.5 Conclusion

In this paper we have presented the results of a case study of several complex industrial systems. We will conclude the paper by providing a synthesis of our interviews with respect to the life cycle of a system. For each phase of the life cycle we will present our main findings.

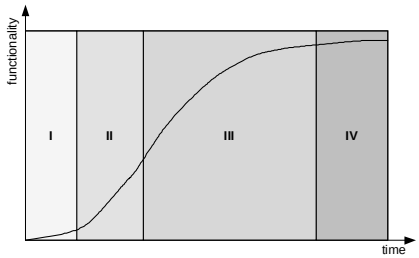


**Figure 9 A system life cycle plotted as functionality over time**

A system's life cycle can be divided roughly into four different phases as depicted in Figure 9: (I) inception, (II) initial development, (III) maintenance and evolution, and (IV) end of life time. The curve in Figure 9 plots the functionality in the system over time. Hence, for a successful system it is desirable to stay in the phase III as long as possible with a curve that has an inclination as steep as possible, because this implies a high degree of productivity. The architecture is the means that should ensure a long life cycle.

It is the initial phase that lays the foundation for a long life cycle. A core team of experienced designers should be responsible for deciding the important architectural principles and the infrastructure. The architectural principles typically manifest themselves as guidelines, handbooks and an infrastructure as a platform. It is the architectural principles that ensure that the most important non-functional requirements are fulfilled. The most common mistake reported is to involve too many people in the initial phase. Several of the interviewees reported experiences in which too many people were involved without having a clear understanding of how to implement what. At best, the development is hesitant and in the worst case, many creative engineers design diverging system components which are incompatible because there are too few constraints on their work. This problem often occurs because higher management requires too much result in too short a time and believes that increasing the number of engineers employed in the project will solve the problem.

It is important that the architectural principles are properly communicated through the development organization in order to preserve them. However, no or limited strategies for communicating the important architectural principles were found in the case studies. One

strategy that was reported as successful was to appoint members of the core architecture team as technical leaders in the development projects. In this way technical leaders become the medium that carries and transfers information of importance to the development organization.

The architects or technical leaders also have a very important role in bridging the gap between architecture and technology, acting as mentors or guides [19]. It is crucial that the architect is a technically very competent person, able to handle both detailed technical issues in the implementation view as well as the coarse-grained big picture represented by the architecture and the domain. Only this kind of architect will be trustworthy in the eyes of the developers who implement the system. Architects should also be able to rise above the small-detail problems and find a better solution on a higher level, when appropriate.

Communicating the important architectural principles is a continuous process that must be considered throughout the life cycle of the system. It is important to define processes and strategies, not only for communicating architectural principles, but also for managing for example, the satisfaction of new requirements. A possible reason for finding process issues rather insignificant could be that engineers who develop systems and those responsible for process related issues tend to have a different focus – product and process quality.

The majority of the systems we have studied are continuously exposed to new requirements from the customers. If these are not properly handled, the system evolution may result in architectural deterioration. As a system is maintained and new functions are added the technical complexity will increase. This is especially true if the architecture does not completely adopt and support the new functions.

Moreover, the cognitive complexity also becomes a problem if there is a large turnover in personnel. Consequently, it is important to have continuity in the people working with the system since they are carriers of undocumented and important knowledge. In the majority of the cases in our study, tools or processes for handling new requirements were not used. As a consequence, the focus during system evolution is on the new requirements only. This creates a risk that old requirements are violated while new requirements are being implemented. Ideally, old requirements should always be verified when new requirements are being implemented. We found no systematic approach to solving this problem.

Finally, the system reaches a point (phase IV in Figure 9) where the current architecture cannot support the new requirements or they become too difficult i.e. too expensive, to implement within the frame provided by the architecture. As the inclination of the functionality curve in Figure 9 decreases, the effort of adding new functions becomes excessive. Paradoxically, this is likely the time at which the company makes the most profit from the system, and will probably continue doing so for quite a while. It is important to set aside funds from that profit for investment in a new architecture. Such an activity should begin towards the end of the phase III.

We believe that the results presented in this paper are applicable in general to complex industrial systems in which software has an important and growing role.

In this paper, we have identified several areas that deserve more attention and each can be the profitable subject of a new study. Additionally, in our future work, we plan to perform additional interviews with smaller organizations and study their systems and compare the results with our current observations and conclusions. We have performed one such interview, which is not discussed in this paper. Before we can publish any results we need to perform additional investigation.

## 2.6 Feedback from the conference attendees

*Comments from a software architect at Nokia, Finland:* "This work reflects quite well our experiences at Nokia. However, in the case of mobile phone products, the HW still tends to come first. One reason behind this is that these affordable consumer products are produced in vast quantities and that the manufacturing costs need to be minimized (in large production numbers the cost of HW dominates the manufacturing cost). On the other hand, as many features are increasingly implemented by SW, the SW is coming more and more important. Another difference is that process issues are important in our case, where for Symbian OS based phones, we have three different product platforms that each supply SW for different families of mobile phone products. The development work is highly distributed (and parallelized) and under heavy time-to-market pressure demanding a high degree of reuse. Without commonly agreed ways of working, SW development would be quite impossible."

## 2.7 Acknowledgments

We are very grateful to the following people who participated in this case study and their companies: Peter Ericsson (ABB Automation Technologies AB/Robotics); Ulf Olsson, Hans Brolin and Mike Williams (Ericsson AB); Nils-Erik Bånkestad (Volvo Construction Equipment), Jakob Axelsson (Volvo Car Corporation); Peter Cigéhn (TietoEnator Telecom & Media); Erik Gyllensvärd and Peter Sandberg (Bombardier Transportation).

# PART II – SYSTEM QUALITY AND OPENNESS IN SOFTWARE ARCHITECTURE DESIGN

> *"In most software applications, investments in software dependability compete with investments in such alternate capabilities as functionality, response time, adaptability, and speed of development. Investigating the tradeoffs among these sources of investment raises a number of significant questions about the nature of software dependability and its interactions with other desired software capabilities."*
>
> *Barry Boehm*

This part of the thesis presents the research results that are used to answer and support question Q2 and hypothesis H2.

Chapter 3 contains publication [38] in unchanged form and is more recent and significant contribution than Chapter 4. Figures, tables and reference numbering have been adjusted to follow the numbering of the thesis.

Chapter 4 is based on the published paper [37]. Because that paper was published relatively early in the research project, certain updates were done to make the paper suitable for the thesis. The updates have been done based on reviewer's feedback and feedback at the conference, and to align the terminology with the terminology used in the thesis. However, the main layout, observations and conclusions are the same.

# 3 A Dependable Open Platform for Industrial Robotics – a Case Study


## Abstract

Industrial robots are complex systems with strict real time, reliability, availability, and safety requirements. Robot controllers are the basic components of the product-line architecture of these systems. They are complex real time computers which control the mechanical arms of a robot. By their nature, robot controllers are generic and open computer systems, because to be useful, they must be programmable by end-users. This is typically done by using software configuration parameters and a domain and vendor-specific programming language. For some purposes, this may not be sufficient. A means of adding low-level software extensions to the robot controller, basically extending its base software platform is needed when, for example, a third party wants to add a completely new sensor type that is not supported by the platform. Any software platform evolution in this direction introduces a new set of broad quality issues and other concerns. Dependability concerns, especially safety, reliability and availability, are among the most important for robot systems. In this paper, we use the ABB robot controller to show how an architecture transformation approach based on quality attributes can be used in the design process for increasing the platform openness.


## 3.1 Introduction

The demands of industry for safety at work and 50.000 hours of mean time between failures (MTBF) require the hardware and software of industrial robot systems to be of very high quality. Industrial robot systems consist of one or more mechanical units, e.g. robot arms that can carry different tools, electrical motors, a robot controller (computer hardware and software), and clients (see Figure 10). Clients are used for on-line and off-line programming of the robot controller.

The software of the ABB robot controller discussed in this paper can be divided into: *platform software*, *application* and *configuration software*. The variations between different products within the product line are currently accomplished through the configuration and application software while the platform software is fixed. The focus of this article is on increasing the number of variation points in the software system through the design of an open software platform architecture for the robot controller. Unless explicitly stated otherwise, *system* in this context is the software system of the robot controller, and *platform*, the software platform of the robot controller. The software platform is the basis for the product-line architecture and its role is similar to that of a domain-specific operating system. One of the

differences between the two is that the platform is less flexible with respect to extension and its share in the responsibility for the final system properties is greater than that of an OS.

According to [24], components in open systems do not depend on a single administrative domain and are not known at design time. As a measure of openness, we use the diversity of the platform extensions and the layer-level in the layered software architecture, in which extensions can be included in the platform. Increasing openness in the platform means, at the same time, that it is possible to increase the number of variations within the product line. An example would be to introduce functionality not available from the base system manufacturer, e.g. new type of sensor in the system. With a closed platform, it is only the development organization responsible for the platform that can add low-level extensions to the system.

When designing the platform for a product line, there are many, varied architectural level aspects that must be considered. Some are of a technical nature, e.g. defining what type of new functionality should be supported and defining the open platform architecture. In addition, many other related non-technical aspects, organizational, business and processes issues are involved. Even though, in this paper, we focus on the technical aspects, each of those different aspects is important. One of the main technical challenges in designing an open platform is in increasing its openness without jeopardizing the quality of the final system. Consequently, all precautions must be taken to maximize the positive contribution of extensions to the platform, and to minimize any negative side effect on the behavior of the final system. As Bosch says [12], “the qualities of the product-line architecture are not relevant in themselves, but rather the way these qualities translate to the software architecture of the products that are a part of the product line”. More specifically, we focus on the following technical aspects of the problem we have described:

- systematic analysis and modeling of the open platform quality constraints, the first step in architectural design, and
- we show how a combination of the fault prevention means and architectural transformations can be used when designing the open platform. The results from the previous step are used for evaluation of the design decisions.

The paper is divided into seven sections. Section 3.2 begins with a short description of the ABB Robotic System and robot controller, the subject of this case study. In Section 3.3, we define *platform openness*. The design approach that we use in this case study is motivated and described in Section 3.4. In Section 3.5 we analyze and model the constraints, quality expectations and the software development architecture, which are the basis for the architecture transformation process described in Section 3.6. Finally, we present certain conclusions in Section 3.7.

## 3.2 System Overview

The ABB robot controller was initially designed in the beginning of the 1990's. It was required that the controller should be capable of use in different types of ABB robots, and thus the architecture is a product line architecture. In essence, the controller has a layered, and within layers an object-oriented, architecture. The implementation consists of approximately 2500 KLOC of C language source code divided into 400-500 classes, organized in 15 subsystems, which makes it a complex embedded system. The system consists of three computers that are tightly connected: the main computer that basically generates the path to follow, the axis computer, which controls each axis of the manipulator, and finally the I/O computer, which interacts with external sensors and actuators.

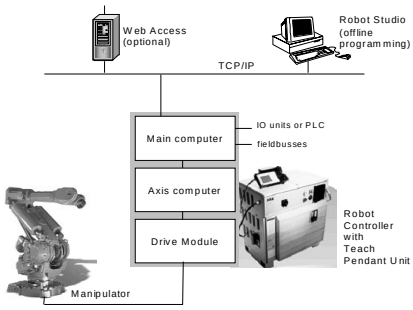


**Figure 10 System overview of an industrial robot system from ABB Robotics**

Only the main computer of the three computer nodes in the original system provides system-users with limited openness. This openness enables end-users to write their robot programming logic in the form of RAPID, an imperative language. A typical RAPID program contains commands to move the manipulator arm to different positions, making decisions based on input signals and setting output signals. This can be done through off-line programming tools on a PC, or on-line programming on a hand-held device designated a Teach Pendant Unit.

The system was originally designed to support easy porting to new HW-architectures and new operating systems. A level of openness beyond that possible with the RAPID language was not initially required. Furthermore, the system was not initially designed to support temporal analysis. Opening up the system for certain low-level extensions, e.g. adding new tasks, would require the introduction of such analyzability.

## 3.3 Defining the Open Platform for the Robot Controller Software System

In this section we define the platform openness from several different points of view. In order to prepare a successful architectural design, it is necessary to understand these different points of view, because all have an impact on architectural decisions.

As shown in Figure 11, we can consider the complete functionality of the system as a combination of functions or services provided by the system in different *system modes* of operation. Some of the modes are associated with normal performance of the system's task, while others represent degraded operation modes as responses to errors in the system. Defining the system modes is one of the steps in defining the *operational profiles* as described by Musa in [36]. According to Musa, system modes are a set of functions or operations that are grouped for convenience when analyzing execution behavior. Specifying degraded operation modes and modes available in different phases of the system's mission is important when system failure assumptions are considered in a system design [25]. The main system modes for the robot controller are the following (illustrated in Figure 11):

- *Initialization mode* – the mode in which the system operates during system startup. This mode is characterized by rapid dynamic changes of the system configuration between stopped mode and normal operation mode.
- *Safe-init mode* - the mode in which the system operates, if the system for some reason, is unable to start in the normal manner.
- *System update and configuration mode* - the mode in which new software may be added to or existing software replaced in the system.
- *Normal operation mode* – the mode in which the system's performs its primary functionality.
- *Fail-safe mode* – if an unrecoverable error is detected during normal operation, the system transitions into this mode. In this mode, the system is optimized for fault-tracing in the system by a maintenance engineer.

The functionality available in these modes, the number of system components and the way the components are connected to each other, are different in the different modes. On the top level, the system can only be in one of these modes at a time. This does not imply that some maintenance functions are not found in, e.g. normal operation mode, but rather that the majority of the functions present in each mode are still related to the primary purpose of that mode in the system.

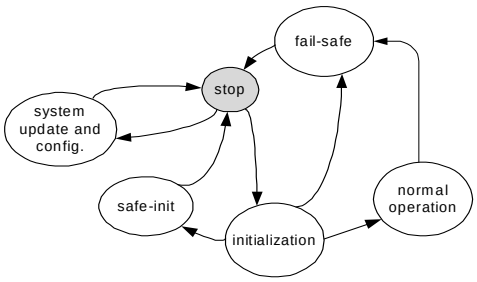


**Figure 11 Functions of the system are experienced by the user as multiple system modes**

The *functions*, also referred to as *services*, that we have described, are implemented either by the platform of the open system, or by components added to the platform. Further, added components should only be able to modify the behavior of a selected number of the platform services as illustrated in Figure 12.

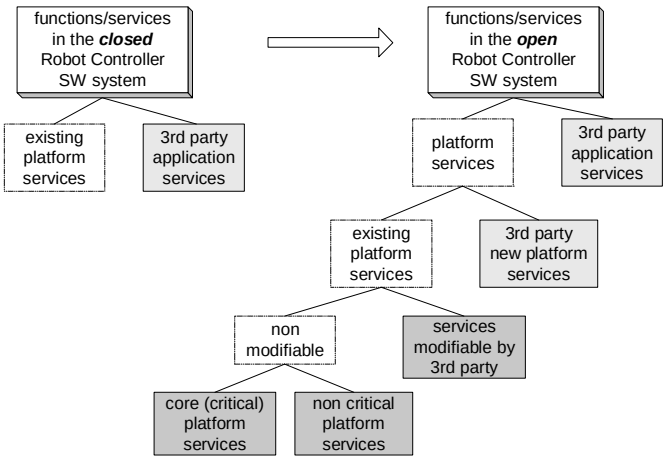


**Figure 12 In a system with an open platform, functions can be a part of the existing platform or extensions to the platform.**

We use the designation platform *extensions* for system components that implement third party platform services or that implement modifications of the existing platform services. We will discuss extensions in two different contexts – the *software development architecture* and the *software operational architecture* of the system.

### 3.3.1 The Software Development Architecture

In this section, we will introduce the terminology used throughout the rest of the paper when we refer to the different components of the final system and illustrate the phases of the system development in which these components are added to the system.

The development artifacts, their relationships and dependencies can also be described as a system architecture as such. We refer to such an architecture as *software development architecture*. As discussed in [6] this architecture could be treated as an architecture on its own, not just as a different view of the single system architecture. In Figure 13, we show components in the software development architecture that can be developed independently:

- *Open Base Platform* – implements existing platform services, which can be modifiable or non-modifiable.
- *Extensions* – implement new platform services or modifications to the existing modifiable Open Base Platform services.
- *Extended Open Platform* – an instance of a platform, which is a composition of the Open Base Platform and all the installed extensions for that particular instance of the platform.
- *Application Programs and Configuration* – application logic written in the RAPID language and application configuration data.

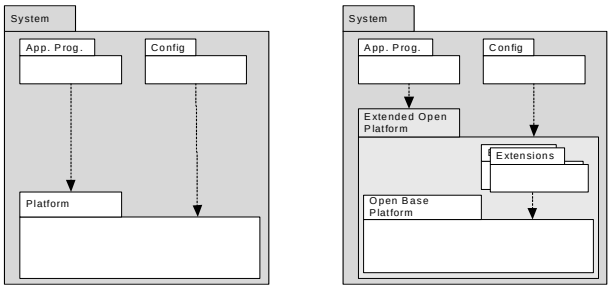


**Figure 13 The number of independently developed packages increases from the closed platform (left) to the open platform (right)**

The left-hand part of Figure 13 shows the robot controller platform without the possibility of adding platform extensions. The right-hand part of the figure shows the open robot controller to which platform extensions can be added. *Packages* represent components that can be developed independently and within different organizational boundaries. A package may contain the implementation of several extension components. The arrows between the packages show their dependencies. When a system in the product line is implemented, the same Open

Base Platform component is always used, while there are many possible choices of extension packages, applications and configurations.

### 3.3.2 Extensions and Software Operational Architecture

While, in the previous section, we showed how different components are related in the development architecture of the product-line system, in this section we describe how platform components fit into the system's operational architecture. Because the system has a layered architecture, we use a high-level layered view to describe it. We say that a system is more open if components can be added to lower layers in its architecture. To make the description more illustrative, we compare the robot controller to an open desktop platform.

#### *3.3.2.1 Open Platforms in the Desktop Applications Domain*

Examples of open platforms are Microsoft Windows© and Linux operating system platforms. Microsoft Windows© is more relevant to our case because we do not discuss open source openness, but rather openness in terms of the possibility of increasing the system capabilities. In the case of Windows© it is possible to extend the base platform on three different basic levels: device driver level, win32 programs and .NET applications. This is illustrated in Figure 14 a.

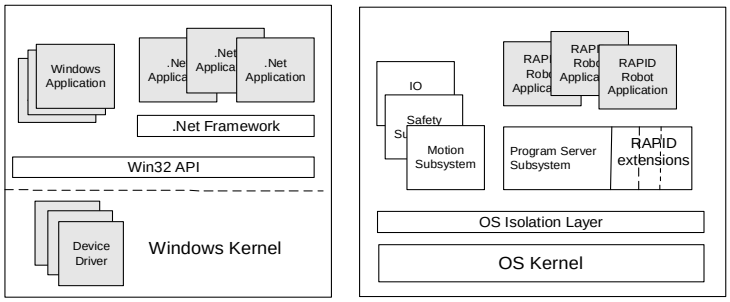


**Figure 14 Different ways to extend Windows© platform a) and ABB robot controller software b)**

The architecture of the system is such that each of the different extensibility mechanisms can only harm the system to a limited degree, device drivers causing the most harm and .NET application the least. Apart from basic or native ways to extend the platform, many of the applications define their own extensibility mechanisms, e.g. Internet Explorer and SQL Server. The differences between these applications are apparent in the process that accompanies the procedure of incorporating the extension functionality in the base system. Scripts are routinely downloaded and run on computers while browsing the Internet, whether requested by users or

not. Application programs are mostly explicitly installed in the system but it is very unlikely that they will cause instability of the system as a whole. In the case of device drivers, a special organization designated Windows Hardware Quality Labs (WHQL), is responsible for certifying such drivers [34]. During the installation of device drivers, users of the platform are clearly notified if the driver has been verified by WHQL or not. The criticality of the device drivers is the driving force for adoption of the latest results from research communities into the device driver verification. An example of this is a recent adoption of the tool Static Driver Verifier, which is based on the model checking research [35]. This tool has a set of rules that can be automatically applied to device drivers, to check if they satisfy these rules.

#### *3.3.2.2 Openness in the Robot Controller*

The current way of adding functionality to the ABB robot controller, by adding RAPID programs, corresponds to adding .NET applications to Windows©. This is shown in Figure 14 b. Two examples of low-level extensions to the robot controller platform that are currently restricted are extensions to the robot programming language RAPID and the addition of new subsystems/tasks. Let us consider an example.

Basic commands in the programming of a robot are motion commands which instruct the robot to move its arms to different positions. Some of the basic motion commands are implemented as RAPID extensions and perform their tasks by communicating with the motion subsystem. The basic part of the Program Server contains the execution engine for RAPID programs, and has features such as single stepping and executing the program backwards, i.e. running the robot backwards. The robot programming language consists of a predefined set of commands. It is desirable that new instructions can be added to permit easier programming, e.g. to permit the easier use of special tools by the robot. Such extensions may require adding new tasks or subsystems which would need an open access to the lower level services in the system. Such extensions may have a significant impact on the system behavior, e.g. the temporal behavior. Traditionally, this has been restricted by the base platform development organization because of the prohibitive costs of verifying that they cause no harm to the system.

## 3.4 Describing the Design Approach

Understanding which types of extensions can be implemented and where they fit in the architecture is not sufficient when reasoning about, evaluating and motivating architectural decisions regarding the open robot controller platform. It is at least equally important to understand the constraints of the environment. Even though this part of the development process is usually considered as requirements engineering, the requirements engineering is closely related to architectural design and the line between the two cannot easily be drawn. As pointed out in [29], the idea that requirements should always state what a system should do rather than how it should do it, is an attractive idea but too simplistic in practice. The goals to be

accomplished and the constraints on the solution must be modeled and described in such a way that it is easy to evaluate the design decision in relation to them.

A naive approach to transforming the robot controller platform into an open platform could be the following:

- Turn internal interfaces into public interfaces.
- Use the public interfaces to develop extensions with off-the-shelf development environments, compilers etc.
- Define how extensions become a part of the platform.

In such an approach, the existing operational architecture is left unchanged. This approach is not acceptable if quality concerns are to be taken into account. In order to come to a satisfactory design solution, we need a design method that explicitly takes quality attributes into account. In the remainder of this section, we will describe several design approaches which focus on quality attributes and the approach we have used in this case study.

### 3.4.1 Different Approaches with Focus on Quality Attributes

In software quality research, different *quality models* exist, e.g. the ISO 9126 model and the model proposed by Dromey [17]. A quality model of a system can be described as a framework to capture the quality expectations of a system. Dromey emphasizes the importance of creating a quality model that is specific to a product [17]. This quality model is designated a *product quality model* and contains a *quality model*, a *software product model* and a link between them, as illustrated in Figure 15. The quality model should reflect the expectations that origin from the specific domain. A refinement of this model is necessary so that it can be linked with the product model. The software product model is the solution side of the model and describes the design or implementation. In the construction of the product quality model, a combination of the top-down and bottom-up approach is used. The quality model is constructed top-down starting from the most general quality attributes of the domain and then refining them to more specific quality attributes. The software product model is more likely to be constructed bottom-up because of, e.g., the use of COTS and legacy components. The quality of the individual components and the ways the components are combined in the product, have a dominant impact on the system quality.

The quality model consists of high-level quality attributes that are system non-tangible properties. Quality attributes can be classified to the following two groups:

- *Operational* quality attributes. Examples of operational quality attributes are: usability, reliability, availability, safety, security, confidentiality, integrity, performance and maintainability.
- *Non-operational* quality attributes. Examples of non-operational qualities are: maintainability, evolvability (reuse over time and across products in a product line), portability, verifiability, integrate-ability and deploy-ability.

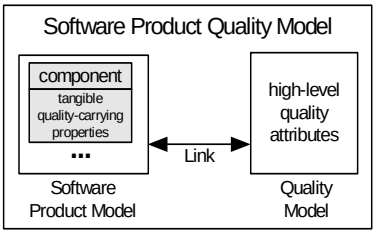


**Figure 15 Dromey's software product quality model**

Since high-level quality attributes are non-tangible and can only be measured through other tangible properties, Dromey suggests that they should be refined to more specific qualities and *characteristics* [18] , in order to create a finer grained quality model. Characteristics are tangible properties of a system that contribute to operational and-non operational qualities. Examples of characteristics are: machine independent, modular, static. Usability can, for instance, be further refined as learnability (easy to learn how to use), transparency (easy to remember how to use), customizability (can adjust to specific needs), operability (easy to access the functionality).

By analyzing the software product quality model, we have come to the conclusion that design tradeoffs between quality attributes are actually tradeoffs in choices of components and different ways of composing the components.

The software architecture community has been very active in trying to find a way to take quality attributes into account during architectural design. Good examples are two recent books on Software Architecture [7,12]. Even though the work described in these books relates to software systems, much of the reasoning can be applied more broadly to computer-based systems. In the architectural analysis, tradeoffs are made between all relevant quality attributes, not only between dependability attributes.

NFR (Non-Functional Requirements) is another approach to dealing with quality attributes which is used within the requirements engineering community. An example is NFR Framework described by Chang et al in [14]. NFR Framework uses a process- oriented approach and a qualitative approach to dealing with NFR, because quantitative measurement of an incomplete software system (its design) is considered to be even more difficult than measurement of the final product. NFR Framework defines soft-goals interdependency graphs for representing and analyzing NFR as well as dealing with tradeoffs.

A generic framework named *dependability-explicit development model* is described by Kaaniche, Laprie and Blanquart in [25]. This framework allows dependability-related activities to be structured and incorporated into the different phases of the *system creation process*. The activities are grouped in four processes: *fault prevention*, *fault tolerance*, *fault removal*, and *fault forecasting*. Because the dependability-explicit development model is a generic framework, customization is probably required for a given system. It is recognized that factors

such as the complexity of the system, the priority of the dependability attributes and the confidence level to be achieved, cost limitations and standards, all influence the selection of optimal solutions.

### 3.4.2 The Approach Used in This Case Study

As explained in the previous sections, the Open Base Platform is the basic building block in the software product-line architecture of the robot controller. The software system and its components are designed and implemented in several stages, and within different organizational boundaries. This needs to be taken into account during the design of the Open Base Platform. In our opinion, the approaches presented in Section 3.4.1 are too general to be directly applicable to our case. For instance, the dependability-explicit development model takes only a subset of the relevant quality attributes (the dependability attributes) into account and gives no support to the development of flexible product-line architecture, which is important in the case of the robot controller. The approach in our case study contains elements of the architecture transformation method and the product-line design method described by Bosch [12], and also elements from the dependability-explicit development model [25]. The approach can be described in the following way:

- We describe the quality goals for the system in terms of constraints and quality attributes organized in two models or views: the operational constraints and quality attributes that capture the dependability objectives, and the non-operational constraints and quality attributes, i.e. qualities related to the development architecture. Quality attributes are organized in such a way that makes evaluation of different design decisions easier. We also describe some important constraints that determine allocation of dependability objectives among the system components. This is further described in Section 3.5.
- We describe and use an architectural transformation-based design method when designing the open platform architecture. In this method, we use the previously defined objectives to evaluate and guide our decisions. The method and its application to the robot controller system are described in Section 3.6.

## 3.5 Modelling Design Constraints for the Robot Controller Software – the First Step in Architecture Design

In this section we present a *quality view* of the robot controller software that includes two dimensions. The first dimension assigns the operational quality attributes of the industrial robot system to the robot controller software system's operational modes. The second dimension is a development architecture dimension that contains non-operational quality attributes. The purpose of these two dimensions is to permit the organization of quality attributes in a structured way that is suitable for qualitative reasoning in architectural design. Such

organization makes the relations between quality attributes more clear, than e.g. having only a flat prioritized list of quality attributes. In a product-line-based approach, it is important to understand how the quality attributes of the platform, which constitute the base-component of the product line, are related to the quality of the products.

### 3.5.1 Operational Constraints

In many software system examples and studies, the analysis begins with software as the top system level. However, when software is embedded in a larger system, a systems approach is necessary when reasoning about quality attributes. In this section, we begin from the quality attributes implied by the industrial robotics domain and derive quality attributes down to the robot controller software system modes.

#### *3.5.1.1 System Level Operational Quality Attributes for Industrial Robotics Domain*

The most important operational quality attributes for industrial robots are the following: dependability attributes, usability (e.g. ease of reconfiguration and operation), and performance (e.g. speed of manipulation, movement precision, and handling weight). For the dependability attributes we adopt the terminology presented in [5]. Dependability is described by the following attributes: *Reliability*, *Availability*, *Safety*, *Integrity*, *Confidentiality*, and *Maintainability*.

*Security* related attributes, i.e. confidentiality and integrity, are usually of less importance for industrial robots as robots tend to be physically isolated, or only connected to a control network together with other industrial devices. However, integrity of data which is not security-related is very important. For instance, it is not acceptable that, e.g. a task in the system should cause a hazard situation by damaging the logic of the safety subsystem. In some cases, integrity of the system is required in the sense that an unauthorized person may not change operational parameters for a robot. All other dependability attributes are highly relevant.

Even though the contact between humans and robots in an industrial environment is restricted (robots work in cells, which are physically isolated by a fence), safety can never be underestimated since an unsafe system can cause considerable physical damage to the expensive robot equipment and its environment. For example, larger types of robots are powerful machines currently capable of manipulating a weight of over 500 kg. Industrial robots are included in the category of safety-critical systems which can be implemented as “fail-safe”, i.e. they have a fail-safe mode.

The industrial/business environment in which robots are used is such that it is crucial to have a very high degree of *availability* and *reliability*. Unreliability leads to non-availability, which means unscheduled production stops associated with what can be huge costs. Because of the complexity of, for example, a car production line, the non-availability of a single robot will result in the stopping of the entire production line. The failure of a single robot with a welding tool caught inside a car body could cause the loss of up to one-day’s production from the production line.

*Maintainability* is important in the sense that it is related to availability. The shorter the unscheduled maintenance time, the higher the availability of the system. Regularly scheduled preventive maintenance cannot be avoided since the robot system contains mechanical parts.

When it comes to the dependability threats, i.e. fault, error and failures, both hardware and software faults must be considered. The robot controller software has both the roles of sending control sequences to the hardware and predicting preventive hardware maintenance.

There are many different fault-tolerance methods that can be applied to industrial robots. However, error recovery with temporary graceful degradation of performance is not acceptable. A robot either works or it does not work; an individual robot cannot perform its tasks by working more slowly - it being only one link in a production chain, or because of the nature, such as arc welding, of the process concerned.

#### *3.5.1.2 Robot Controller Software System Operational Qualities*

In Figure 10 a typical robotics system and its subsystems is shown. The system architecture is such that the domain dependability quality attributes discussed in Section 3.5.1.1 do not affect all of the subsystems in the same way. How domain requirements are propagated to subsystems may differ between different robotics systems depending on their design and implementation. When propagated from the system level, quality attributes required of the robot controller software subsystem can be described as following:

- *Safety* – The robot controller has software logic responsible for the collision detection safety feature.
- *Reliability* and *availability* – The robot controller is responsible for generating correct coordinates (values), and with predefined time intervals so that signals can be constantly sent to the manipulator while it is moving. If incorrect values are sent, the manipulator will move in the wrong direction and if no values are sent, the manipulator will not know in which direction to continue its movements and it will stop in the fail-safe mode.
- *Integrity* – Configuration data integrity during maintenance and integrity of data structures in normal operation is very important because of its large impact on reliability and availability.
- *Performance* – Examples of important performance values for a robot are speed of manipulation, and repetitive and absolute accuracy. In software these issues relate to computational efficiency and data transfer efficiency.
- *Maintainability* – In the case of failures, it must be easy to identify and recognize the faulty component in the system and replace it. The manipulator, which is an electro-mechanical device, must be maintained on a regular basis as it is subject to degradation faults. Because of this, any preventive, corrective or perfective updates to the robot controller software and hardware can be done at the same time without affecting the complete robot system availability.

For the purpose of a detailed design, these quality attribute descriptions must be defined more precisely but they are sufficient for the purposes of this paper.

#### *3.5.1.3 Operational Qualities for the Robot Controller Software System Modes*

Different robot controller software system modes, discussed in Section 3.3, are likely to have different or differently prioritized operational quality attributes. The goal of this section is not to go into the detailed design of each of the modes, but to model quality attributes expectations on the functionality in the modes. In Figure 16 we visualize how quality attribute requirements described in Section 3.5.1.2 are propagated to the individual system modes, resulting in the lists of dependability goals to be satisfied in the different modes.

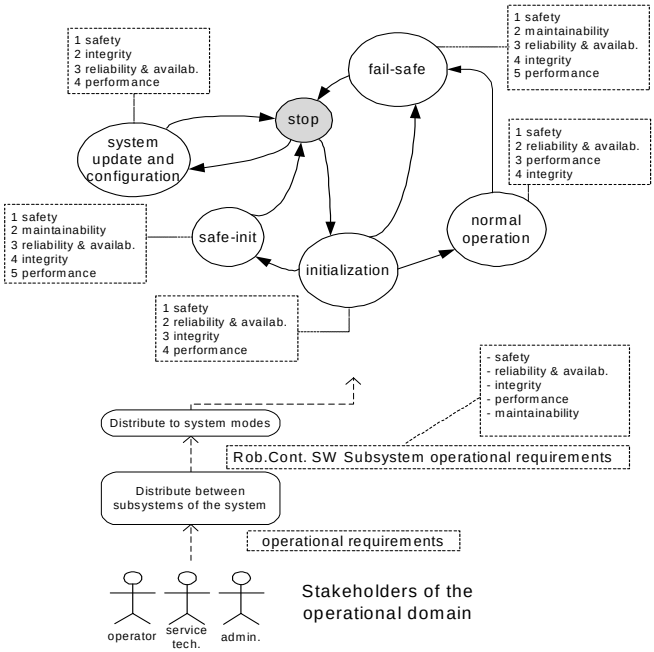


**Figure 16 Operational system modes with quality attributes assigned to them. As development stakeholders are not present here, the model is not dependent on their concerns.**

As mentioned in Section 3.3, these modes have significantly different numbers of components and their interconnections. Because the normal mode of operation is the mode of the system's primary purpose, it is easy to neglect the other modes. From the software system quality perspective, it is absolutely crucial that the transitions between the modes are predictable and reliable. If the software system is corrupted on this level, it may not be able to start again.

This implies that reliability and integrity are very important attributes at the top software system level.

#### *3.5.1.4 New Operational Constraints for the Open Platform*

For the open platform, new operational constraints also exist which are not necessarily of quality attribute type. An example is constraints on the allocation of dependability objectives that specify allowed effects of third party extensions to the core non-modifiable open platform functionality. Examples of such constraints are:

- The safety feature "collision detection" is a core non-modifiable feature allocated to the Open Base Platform components only, and can not be affected by platform extensions.
- The integrity of the top level software system mode transition sequence is allocated to Open Base Platform components only and it must not be affected by extensions in such a way that for example, the system is unable to start.

### 3.5.2 Software Development Architecture and Constraints in the Development Domain

In this section we discuss the non-operational quality attributes that are important in the design and implementation of a system in the product line. In general, a system must be designed to fulfil both the operational requirements and the development requirements. Unlike dependability attributes, generally accepted definitions of which exist within the research community, there is a lack of generally accepted terminology and definitions for other quality attributes. For the quality attributes we refer to, we use intuitive names to refer to their meaning.

To identify the non-operational quality attributes, we first identify the *development stakeholders* and then bind their concerns to the packages in the development architecture that we introduced in the Section 3.3.1. It is generally recognized that quality concerns are dependent on the stakeholders and their views. The way that stakeholders are identified is still, to a large extent, an ad hoc procedure. A recent attempt by Preiss to systematize the stakeholder identification is described in [49]. Stakeholders are divided into *goal stakeholders* and *means stakeholders* and also according to the product lifecycle phases (*development stakeholders* and *operation stakeholder*). In this paper, we discuss only the *means* stakeholders, those who are responsible for designing and building the system. The concerns of the *system operation stakeholders* were described through the domain and system analysis of the robotic system in Section 3.5.1.

The important development stakeholders for industrial robotics and their concerns are presented in Figure 17. Using the terminology from Section 3.4, some of the concerns would be expressed as characteristics, e.g. standard compliance, and some as non-operational qualities, e.g. reuse. In Figure 17, we show only direct concerns of the stakeholders, but concerns of all other stakeholders are indirectly propagated to the Open Base Platform developer. The more effort invested in the design of the base platform level to address these concerns, the greater is

the probability that the quality of all the components of the system will be improved. This is one of the very important goals of our design work.

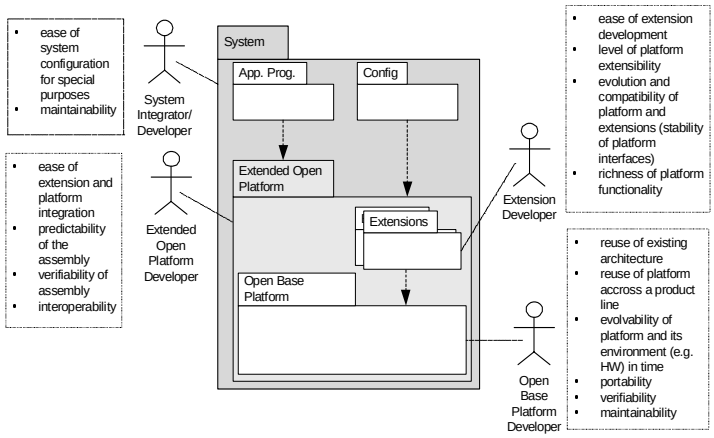


**Figure 17 Software development architecture of the robot controller software system with *means* stakeholders and their direct (first-level) concerns.**

### 3.5.3 Other Important Requirements and Constraints

Operational and development qualities do not give us a complete model of the constraints on the software system design. There are other important constraints that can have a significant impact on the system design. The major tradeoffs in large and complex systems are usually associated with economics and business goals [7]. While we describe none of these constraints, they should definitely be defined and taken into account in the design phase. In the terms of means and goals stakeholders, mentioned in Section 3.5.2, these constraints are related to the goal stakeholders.

## 3.6 Robot Controller Software System Architecture Transformations

In this section we describe the design process for a more open software product line, which includes the architectural transformation process for the Open Base Platform. Referring to Figure 17, we are in the position of the developer of Open Base Platform. Our design goals are the following:

1. Direct goals from the Figure 17, such as reusing as much as possible of the existing assets, including the architecture.

2. Indirect development goals from the Figure 17, which are to help other development stakeholders to meet their goals. If extensions are difficult to develop and integrate into the Open Base Platform, it decreases the likelihood that many new extensions will be developed.
3. To meet dependability objectives allocated to the Open Base Platform operational architecture components. These objectives were described in Section 3.5.1 through the domain analysis and the assignment of dependability goals to system operational modes.
4. To provide fault prevention means in the form of software tools, to other development stakeholders, in order to help meet the system dependability goals. This goal is closely related to item 2 in this list. If the complexity of the system integration and extension development is high, it increases the likelihood that the reliability of the system will be lower.
5. To meet other constraints, e.g. business goals discussed in Section 3.5.3.

These goals are independent of whether there is or not an existing robot controller software system. As we have an existing system from which to begin, we can describe our design goals in a slightly different way – we need to transform the system to address the changed and new requirements while still fulfilling the requirements that have not changed. From the discussions in the previous sections, we obtain the list of changes in the system and its environment, which can be attributed to the increased openness. This list is presented in Table 1.

**Table 2 Changes in the robot controller software system and its environment, which can be attributed to the increased openness in the platform.**

| | |
|---|---|
| 1 | Some of the system *characteristics* have changed:<br>• The amount of non-critical code in the system is very likely to increase. This increases the overall system complexity and the likelihood of faults being present in the system.<br>• Adding extensions to the system increases resource utilization in the system which increases the risk of the system running out of critical resources during operation. |
| 2 | The system development environment has changed significantly, as shown in Figure 13 and Figure 17. Concerns of the new stakeholders, i.e. developers of the Extended Open Platform and developers of extensions, need to be addressed by our design. |

| | |
|---|---|
| 3 | New operational constraints on allocation of dependability objectives between the system's components have emerged (described in Section 3.5.1.4). |
| 4 | Other new objectives exist, e.g. business objectives, which are not discussed in this paper. |

To design a system, which is perceived as a high quality system from several different perspectives, all of these new goals must be addressed. Dependability of the system during operations can be categorized as a subset of these goals. System operational qualities as described in Section 3.5.1.3 are independent of the level of openness in the system. From the operational perspective, who has designed and implemented a component is irrelevant. However, this does not mean that there is no link between the development process/architecture and the operational architecture. The relationships between the different artifacts of the architectural design are visualized in Figure 18.

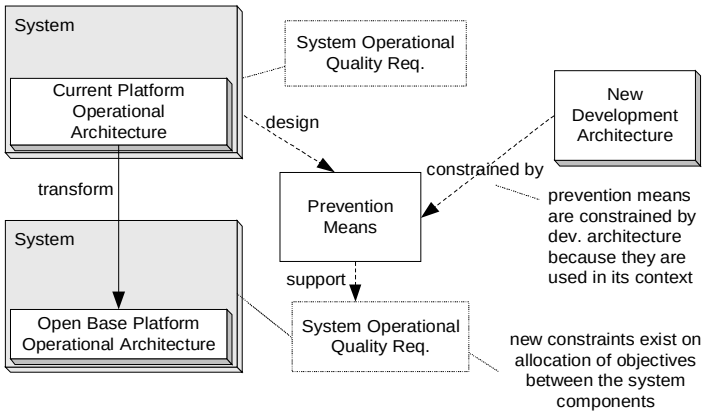


**Figure 18 Relationships between the software architectural design artifacts**

The current operational architecture of the platform is transformed into an Open Base Platform operational architecture. The platforms are the basis for systems that have the same operational quality attributes. However, for the more open system, new constraints exist on the allocation of dependability objectives between the system components. Fault prevention means,

such as evaluation of the properties of the composition of the base platform and extensions, enforcements of restrictions on programming language features, are used to achieve the quality goals of the final system. The development architecture is changed and it dictates how and when fault prevention means can be applied.

### 3.6.1 Architectural Design

It is known that it is hard to define what software architecture is [6]. Furthermore, there are differences between what is considered an architecture in industry and in the research [20]. Fowler argues that all important system level design decisions that are hard to change are architectural decisions [19]. What is considered to be important is subjective and dependent on the opinion of the system expert.

One of the findings from our recent case study involving architects from seven large industrial companies was that an architects awareness of the process that is used to develop a product should be increased and process issues should be taken into account in architectural design [40]. Since the Base Open Platform is the basis for all products in the product line, it must provide a set of tools that can be used to support the process of completing the design and implementation of the system. An example of such a tool is a tool that checks that extensions to the platform are designed in such a way that they conform to the architectural principles of the base platform as described in Section 3.6.3. This is the design level support for operational quality goals of the system. We strongly believe that it is the software architects who should decide: which tools are suitable, what design properties of the system should they check and what are their limitations. In that sense, these tools become a part of the software architecture on the system level.

When designing a software architecture, Bosch [12] suggests preparing a functional design first and then performing architectural transformations to take quality attributes into account. Transformations that may be required could be as simple as adding a new functional component to the system, or may require changes such as the use of different architectural styles or patterns. This depends on the nature of a quality attribute. In our case, the functional design is already available because we began from an existing system. Furthermore, the existing system supports operational quality attributes as described in Section 3.5.1. The list of changes in the system environment was presented in Table 1. Design transformations must be performed to meet the new constraints within the context of the new development architecture.

The outline of the software architecture design method for the transformation of the system with the closed platform into a system with the open platform is shown in Figure 19. Initially, the open platform architecture is the same as the existing platform architecture. A number of architectural patterns in the existing platform architecture already provide a good basis for openness, such as: broker pattern, publish-subscribe and the layered-architecture. Then we estimate if the quality attributes and the constraints are satisfied or not. If not, we consider if prevention measures can be successfully applied to satisfy our concerns. If such is the case, we need to add the appropriate support, in the development environment, e.g. tools, language

restrictions. If prevention measures are not sufficient, then we need to apply transformation of the platform operational architecture, e.g. to apply additional fault tolerance measures. Using prevention measures as much as possible helps to reuse the existing operational architecture, i.e. minimize its modifications, one of the goals mentioned in Figure 8. The resulting architecture is designated Open Base Platform operational architecture.

The platform transformations are performed now, but there is no complete system at this point. Many different systems can be designed using this platform as a basis. The platform design is completed when extensions are added to the platform, resulting in an Extended Open Platform. In the design of the Open Base Platform architecture, the effects of adding extensions to the platform are evaluated qualitatively, but the final configuration is not known. Evaluating the effects of adding extensions, e.g. new tasks, requires a tool support that can be used in a simple way outside the organization that designed the platform.

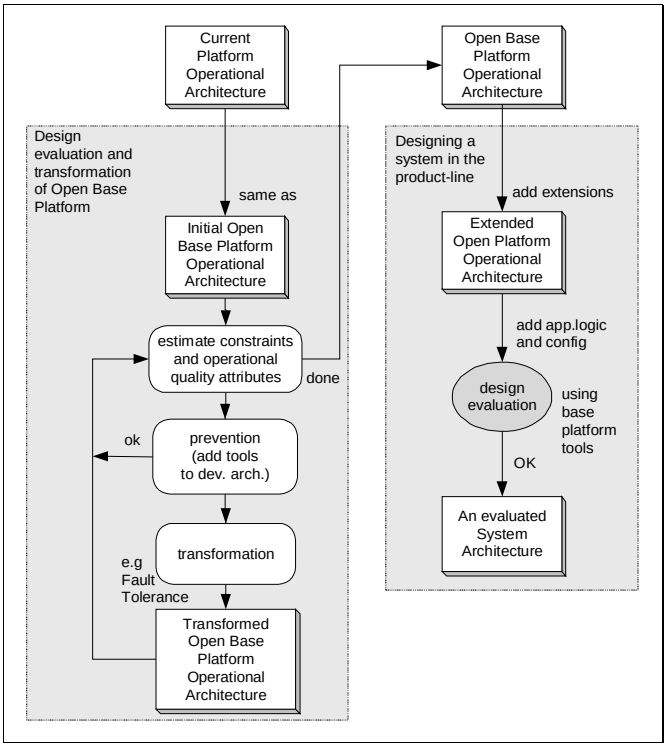


**Figure 19 Architectural transformation of the closed robot controller platform to an open platform. The open platform architecture is the basis for many different systems in the software product line.**

The following list summarizes the dependability means techniques that are applied in the platform architecture transformation phase, to address threats raised by the items in Table 1:

- Fault Tolerance techniques
  - Because of the increased amount of non-critical code in the system (item 1), the system must be partitioned in several *fault-containment zones* representing multiple levels of criticality. The same technique addresses the concern introduced by item 4, on allocation of dependability objectives between the system components.
  - Adding new components increases resource utilization and requires that *watch-dog* techniques be used to monitor the usage of critical system resources. Since the robot controller system may have fail-safe behavior, the watch-dog initiates roll-forward-recovery to the *failure* system mode.

- Fault Prevention techniques
  - When new components are added to the Open Base Platform, the architecture of such a new system must be re-evaluated for important properties such as timeliness. Qualitative reasoning about such properties is difficult. We have therefore introduced a software tool for a quantitative evaluation of the system design that checks for errors such as the violation of deadlines and queue underflow or overflow. Normally, the queue containing coordinate references must never be empty. If, however, it does become empty, the robot does not know what to do and will transit into a fail-safe state.

- Fault Removal techniques
  - When components of a system are developed by different organizations, *fault diagnosis* that records the cause of errors in terms of both location and type is important. Fault diagnosis is the first step in fault handling [5]. However, in our case the goal is not to perform other steps in fault handling, e.g. isolation, reconfiguration, reinitialization, but to give precise information to an external agent. Monitoring and debugging in the distributed real time system is challenging. In [59], Thane describes a method for software-based replay debugging of distributed real-time systems. A case study of the use of this method for the ABB robot controller is described in [60]. When this kind of diagnostic component is introduced in the Open Base Platform, it works in the same way for all components in the system.

### 3.6.2 Fault Tolerance Design Techniques to Support New Tasks

In addition to the architecture level transformations to support extensions to the Open Base Platform, lower level design transformations and techniques are needed to support different

types of extensions. There is a difference between providing support for the addition of a new task to the system and adding a module that implements a new RAPID language command. The impact of changes to the system to support these different types of extensions is local and we therefore treat them as design level changes.

There are three important issues when transforming the architecture to support the addition of tasks, in addition to basic middleware broker support for installing tasks. These are:

- Which interfaces should be provided?
- How is the added task supervised during operation to detect erroneous behavior?
- Which type of exception handling should be used?

The basic system includes a set of interfaces for different required functionalities. The interfaces provided can be classified as critical or non-critical. The critical interfaces provide access to functionality that can impair the basic operational behavior of a robot, such as motion control. A failure in a task that is classified as critical will lead to an emergency stop. A task classified as non-critical is a task that only makes passive use of the information in the system and in the event of a failure, will not require the immediate stopping of the robot. An example of such a non-critical function is a web-server that presents certain statistics from the operations. Other fault handling measures can be taken instead, such as disabling the functionality provided by the extension. An added task that uses interfaces of both classes is classified as critical.

Supervising the behavior of an added task is supported in two ways, (i) assertions provided in each of the services and (ii) supervision of the assigned CPU bandwidth. The CPU-bandwidth is supervised by extending the basic operating system with functionality for monitoring execution time of each added task during each cycle of execution (from the time a task becomes ready until it will be waiting for the next cycle of execution). The basic idea is, thus, to prohibit the added task of delaying lower priority tasks longer than specified off-line. If the task uses too much CPU-bandwidth, different measures will be taken depending on the criticality of the task, as mentioned above.

In the case of a critical task, one can use an exception routine that is invoked before the severe impact occurs and allow that exception routine to finalize the computation with less accuracy. This concept is called imprecise computation or performance polymorphism as suggested in [31] and [58]. However, this approach is not acceptable in our case, since performance polymorphism can yield an application that delivers uneven quality. In the case of a non critical task we can adopt such an approach. In the first version of the design, a controlled shut down of the task is performed and an error message provided to the operator.

### 3.6.3 Software Architecture Tools for Evaluation of Real-Time Properties

As shown in Figure 19, the evaluation of the Extended Open Platform Architecture is performed by other than the Open Base Platform developers. In [7], when the authors discuss product-line architectures, they point out that the documentation needs to explain valid and invalid variation bindings in a system based on the product-line architecture. We find this

necessary, but not sufficient in our case; a tool support must also be provided. It is difficult to evaluate the impact of the combined extension on the temporal behaviour of the system, because of the system size and the number of interdependencies. A robot control system is a composition of Open Base Platform, extensions, configuration options (e.g. different field busses, the robot model, different software options such as spot- or arc- welding) and the user-programmed behaviour in the RAPID language. System compositions resulting from these components have different timing behaviour but in some cases, the magnitude of these differences might be insignificant. With extensions, the base system must not only be verified in different configurations, but also in combination with extensions that are to be used. These extensions can potentially have a large impact on the system timing and the developers of the extended base platform should not be required to know the details of the internal structure of the system in order to estimate the extension impact on the system. A way to address this issue is to describe system components using models and then analyze the system properties based on these models, as shown in Figure 20.

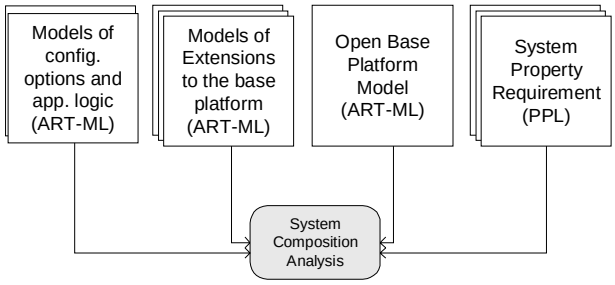


**Figure 20 The ART framework when used for real-time property validation in the design phase.**

We propose a set of tools and methods for analyzing the impact on timing properties when changing the system. We call this framework ART [62]. It provides means for creating, analyzing and validating models of real-time systems. As illustrated in Figure 20, the general idea of this framework is to create a model of the system components using the modelling language ART-ML and, by using a suitable analysis method, to determine how the timing is affected by a change in the system. The analysis tool takes a system model containing the base system and extensions as input. It also needs a set of system-timing properties to analyze, formulated in the probabilistic query language PPL. In the current version of the framework, simulation based analysis is used. The tool evaluates the specified properties against the model.

## 3.7 Conclusions

Making changes to existing complex systems is never an easy task. These systems are created and fine-tuned using years of experience and maintenance, to satisfy the demanding

functionality and quality requirements of the domains in which they are used. When changes are considered, it is vital to understand the impact of those changes, and then to find or develop adequate techniques to address the new or changed concerns and expectations.

In this paper, we use ABB robot controller, a complex real-time system, to analyze the impact on the system when the openness of the system is increased. As pointed out in [49], many development projects primarily take into account the operational domain stakeholders and their concerns. From the broader quality point of view when analyzing the impact of increasing system openness, we come to the conclusion that the impact on the system is primarily on the development side. Further, we recognize changes in some operational characteristics of the system that may affect system operational qualities and dependability. The amount of non-critical code in the system increases as well as resource usage. Because system reliability is a vital property of the system, and because the use of fault-tolerance means supporting fail-operational semantics is limited (because of cost or complexity), the importance of fault-prevention is emphasized, and tools supporting evaluation of the design decisions are proposed. The separation of system components into critical and non-critical components and their isolation in fault-containment regions is used to support fail-safe operational semantics.

We believe that most of the reasoning in this paper is applicable in general to the robotics domain, but also partly to other systems in the embedded domain, which require openness in the sense described in this paper.

# 4 Propagation of Quality Attributes in Layered Architecture – a Case Study in Industrial Robotics


## Abstract

Many software systems are open in the sense that their basic functionality can be extended or modified by adding software components from third parties. An end-user always experiences functionality and quality that is the result of the assembly of all parts integrated into the application, regardless of the source of their development. In the case of functionality, it is often much easier to pinpoint the contributing components, than in the case of quality.

In the case of business critical applications, the quality is one of the primary concerns. In this article, we analyze the relation of system qualities and architectural decisions for an open application, in which dependability quality attributes are very important. The analysis is done from viewpoint of the developers of the open application. Some of the important architectural decisions that affect all components in the system are choices of underlying technologies. Based on the layered architectural style, we apply a quality attribute based model to help us do reasoning in a systematic way. The model is validated through real world experiences from the development of open graphical user-interface application on a device used in industrial robotics.


## 4.1 Introduction

If an Internet browser does not support certain type of content you may be asked to install an extension that handles this type of content. While viewing this new type of content, the browser may suddenly stop responding. Who is to blame? Because quality is such an elusive target [46], the answer to this question will vary a lot depending on the user's background and experience. Many users will certainly blame the browser, because it is the one who stopped responding. The browser manufacturer was faced with a difficult tradeoff. Opening possibilities for programmability, extensibility and innovation by third party increases likelihood of a product being designed for a specific use and therefore more useable. Limiting openness or disabling extensibility mechanisms, makes it easier to improve the dependability quality attributes of the closed system. Level of extensibility for third parties can also be viewed as a measure of system openness.

In this paper, we analyze the role of developers of an open business critical application that is similar to the role of the developer of the Internet browser application, but the consequences of the quality problems are much higher. In the initial design phase, an architect of an open application needs to make choices of underlying platform, technologies, etc and give estimates on the cost of solution and the quality that can be achieved. The open application, in our view, may be a complete application, which allows inclusion of new components, or it can be a

platform that provides a basic functionality and enables inclusion of other applications, or similar. The main point is that a third party can update the application independently of the organization that developed the core application.

The problem of predictable openness is, in a nutshell, the problem of trusted components and their assembly [21]. Voas suggests [61] that certain techniques, methodologies, tools and processes need to be employed to achieve particular level of quality attributes in a product. We recognize the importance of all of these measures, but within the scope of this paper we focus only on the architectural and design techniques. The developer of the extensible application is responsible for setting and enforcing the quality goals by means of these aspects of application development.

We show how quality modelling can be used to reason about quality in a systematic way and also be used in a simple and intuitive evaluation process of the design against quality goals. In our case study, we consider development of an extensible application that runs on a display-based device that is used to program and control an industrial robot. The application fully controls the device display, but it also makes it possible for third party developers to customize this user interface with new elements. The reason for this is to make the device more suitable to the specific process that robot performs, e.g. a certain type of laser welding. A simple layered architecture is used in this application.

The rest of the paper is structured as follows. In Section 4.2, we discuss a quality model based on the layered architectural style. In Section 4.3 we illustrate the application of the model on the handheld device for programming industrial robots. We present conclusion and summary in Section 4.4.

## 4.2 Quality Modelling and Layered Architecture

### 4.2.1 Software Product Quality Modelling

Software Quality has been a subject of research for the last few decades, and in last decade, lots of work has been done related to quality attributes and their influence on the overall quality. Some recent approaches suggest that the top-level quality attributes should be observed and analyzed through more tangible properties that contribute to the overall quality. Dromey describes this approach in a series of [16,18]publications [17] [16,18]emphasizing the importance of building a quality model that relates high-level abstract quality attributes with concrete, tangible quality-carrying properties on the component levels (see Figure 21).

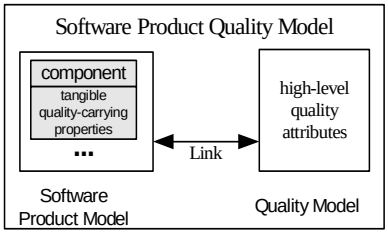


**Figure 21 Dromey's Software Product Quality Model**

There are many challenges in creating such a model. Creating a quality model itself is not an easy task, and creating a link to the *software product model* is even harder. Voas points out that there is not enough validated data available to be able to tie the cost of using a particular technology to the level of a particular "ility" that was achieved [61].

### 4.2.2 System Quality Attributes vs. Component-Quality Attributes

Quality attributes are usually analyzed and determined on the system level. However, when we build open or extensible systems, or in general component-based systems, we must relate the required system quality attributes to the quality attributes of a basic platform or in general of involved components. This relation is not trivial; it depends on many factors and is different for different attributes. One common strategy in analyzing system properties is to use a top-down approach. For open systems, we must combine this approach with a bottom-up approach, when we start with attributes of existing (sub)systems, and propagate them to a (not yet existing) system level. Depending on the types of attributes, and the constraints related to openness, we would be able to reason about the system attributes and achieve a certain level of predictability of the system behavior.

### 4.2.3 Propagation of Qualities in Layered Architecture

Relations between system quality attributes and components attributes are determined by the selected architecture. Different architectural styles enable openness. In our case the layered architecture has been used. In such a case, we must identify quality attributes on different layers and find relations between them. A generic model that shows the propagation of quality attributes through the layers is illustrated in Figure 22.

The lowest layer is the operating system (OS). The OS has certain functionality that applications can be built on. Often, it also includes a support for building new functions with a set of tools and technologies. Quality attributes QA1 are associated with the OS functionality, but also with the tools and technologies.

The design of the application is probably more influenced by the technologies and tools than the functionality of the OS. The choice of tools and technologies will have significant impact on the quality attributes of the application. Efficient and well-designed tools can have very

significant impact on the quality of applications. This is one of very significant elements to be considered in the choice of platform, where platform refers to a bundle of the OS, tools, techniques, documentation, etc. This does not imply that all element of the platform come from the same manufacturer. In some cases, the set of tools and technologies is not OS specific, which is helpful in the development of portable applications.

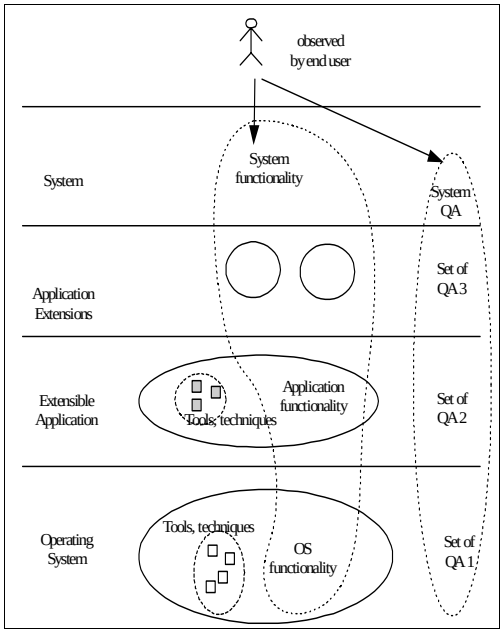


**Figure 22 Model of quality propagation in a layered architecture**

Extensible applications have their own functionality and set of related quality attributes QA2. This functionality may be such to restrict some of the OS functionality, or to enhance it. Quality attributes are supported by the underlying OS qualities, but may be weighted differently than in the OS layer.

In the case of a widely used underlying platform, like Microsoft Windows, the developers of application extensions are often familiar with the platform, its tools and technologies. If these tools and technologies are to be used, all restrictions imposed by the extensible application architecture need to be enforced by additional tools, rather than descriptions and guidelines. Quality attributes QA3 are associated with different components.

Emphasis on the individual quality attributes in the set QA of the system is most likely very similar to the QA2. Developers of the extensible application need to design and implement enough support to ensure the system QA.

## 4.3 Case Study - FlexPendant Device for ABB Industrial Robots

The case study in which we build a product quality model is an analysis of the design of an extensible user interface application on a display-based device. This device is called GTPU (Graphical Teach Pendant Unit) and is used for robot programming and control. Figure 23 shows a system overview and place of GTPU device within the system.

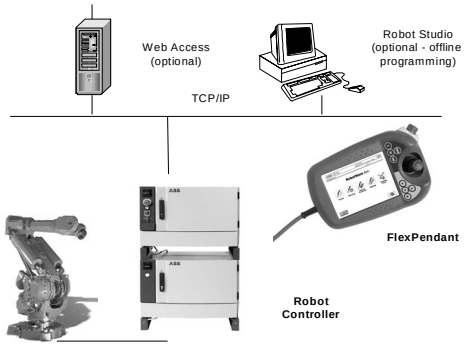


**Figure 23 System view is necessary for understanding the high-level quality attributes of the FlexPendant device and its software**

We start the analysis top-down, by identifying high-level quality attributes of FlexPendant device. To do this we first need to identify the most important stakeholders and also propagate allocated quality requirements from the larger system, which is the complete robotics system shown in Figure 23. In this paper, we identify the most important and most obvious stakeholders: the end-user, the developer of extensions, and the developer of the extensible application. A more detailed discussion on a systematic stakeholder discovery and classification can be found in [49]. From the system end-user perspective, the following high-level quality attributes are the most relevant for the FlexPendant device software:

- Reliability (continuity of services);
- Availability (readiness for usage);
- Usability (easy to use);
- Performance (response time).

The developers of extensions for open FlexPendant device want to ensure that overall reliability, performance and availability in the system cannot be jeopardized by mistakes in the third party extensions. On the other hand, the developers of extensions will be very concerned with extensibility, maintainability of their components, and the amount of effort to create their

extensions. Indirectly, the system that is extended and customized for specific industrial process will most likely be experienced as more usable by the end user.

### 4.3.1 Quality Attributes of the OS/Platform Layer

In the case of FlexPendant, Windows CE was selected as the operating system. The following list of Windows CE QoS quality attributes and their sub-attributes is adopted from [33]:

- Modularity – does not refer to the design time modularity, but runtime modularity in the terms of process isolation and memory protection. We can add that the system is also modular in the design time, so that OS components can be included, or excluded from the OS image. This will not necessarily reflect in the change of runtime/operational modularity.
- Scalability – refers to the good support for managing large number of processes and threads.
- Predictability – refers to OS real-time capabilities, interrupt latency handling etc.
- Adaptability – ability of OS to be extended and enhanced. Adaptability is supported by the design time modularity.
- Stability – refers to the OS stability in long-term usage and stress scenarios.
- Security – refers to protection against malicious and unauthorized access.
- Accessibility – openness of OS functionality and services through various programming interfaces.
- Testability – does not really discuss testability (properties of the design and implementation, which make it testable), but rather describes testing procedures used in the product development.
- Performance – refers to the tuneable and configurable design and implementation that can be optimized according to the specific system needs. However, no details are provided to support this statement.
- Survivability – refers to the sophisticated power management and support to handle power failures.

Note that these are the QoS quality attributes, or also called attributes discernable at runtime. We can notice, that names in this list are not always intuitive and present in the quality model standards. Also, it is questionable that that e.g. accessibility and testability are QoS properties, as in the form they are defined above.

Now let us consider examples of tools and technologies that will have significant impact on the application quality. Windows CE platform provides developers with two different APIs (application programming interfaces) - .NET Compact Framework API (.NET version for

devices; further in text .NET CF) and win32 API. According to [63] the following is list of properties of the named two APIs that will reflect on the application quality.

Advantages of the .NET API:

- Managed code cannot have bad pointers.
- Managed code cannot create memory leaks.
- Managed code supports strong type-safety.

Advantages of the win32 API:

- Fastest executables
- Better real-time support than .NET
- Source code (inter-platform) portability
- Ability to wrap COM for access by .NET CF applications
- Ability to create device drivers
- Ability to create control panel applets
- Support for custom user-interface skins
- Support for security extensions
- Ability to build Simple Object Access Protocol (SOAP) Web Servers
- Support for Pocket PC shell extensions
- Ability to use existing Win32 code

Let us see how some of these quality-carrying properties can help us in the design of our extensible application. However, some of them will not really hold for our case, and there are many more additional quality issues that are not mentioned above.

### 4.3.2 Qualities in the Extensible Application

Having in mind the top-level qualities of the system, we need to select the platform API that our extensible application will be based on. The selection will be made based on the contribution to the qualities on the current and the next layer. In this case, reliability, ease of development of extensions, and maintainability will guide us to choose .NET API. Besides the quality carrying properties mentioned in the previous section, we need to add the following properties:

- Superior tool support. The developers of the extensible application and the developers of extensions can use the same tools.
- Code isolation functionality in the form of application domains
- Good support for versioning
- In-line with the manufacturer’s strategic directions

There are some drawback issues related to the chosen technology that may not be obvious and that should be taken into consideration:

- Missing features – .NET API does not support all features of the underlying platform accessible through the win32 API. At the same time interoperability between the .NET and win32 code in the .NET CF is limited and complicates the design in the .NET code. This will primarily have negative impact on maintainability, testing of our application.
- Memory leaks – Memory leaks can happen in .NET applications that use interoperability with win32. As stated above, this may be necessary to reach unexposed functionality. Also, releasing unused resources even if they solely reside in the managed code is recommended. Performance may suffer from inappropriate use of object allocation.
- Application domains – This is a .NET functionality that can be used for isolation of non-trusted code from trusted code. This element of design will have significant positive effect on the reliability of the system, as it prevents propagation of failures. This feature has significant limitations in the current .NET CF implementation. There is no support for the communication between application domains. This support is needed to do any useful work.
- Exception handling – Exception handling is a powerful technique for implementation of fault-tolerance and, in .NET, it can be used across programming languages. However, we advise more discipline on usage of exception handling, because the .NET CF allows too much freedom that can turn into a problem late in your implementation phase.

The information presented above is just an example of very useful and essential information that is necessary to establish links between a quality model and the software product. Good sources of such information are often communities of the product users, but not the manufacturers themselves.

A developer of an extensible application would like to impose some constraints to the extension developers. An example of such a constraint is that extension developers cannot access win32 API through the interoperability mechanism. A tool can be created to inspect .NET code for violation of such a constraint.

### 4.3.3 Qualities of Application Extensions

The developer of application extensions will have the same set of .NET tools as the base application developer, plus those few additional tools that set more constraints. One significant factor that will influence the quality of the developed extensions is also a good and unambiguous understanding of the underlying APIs and infrastructure. A description of these

semantic issues can be found in [8]. This can be combined with a tool that will restrict the API. Its usage will guarantee some system properties and increase the development efficiency.

#### 4.3.4 Qualities of the Resulting System

.NET CF has been used by ABB, in the GTPU project, since the earliest beta programs for .NET CF development partners. Our experiences gathered so far have shown that choice of the .NET API has a positive impact on important high-level quality attributes reliability and availability. This can be stated for both base application and extensions, as a number of extensions have been developed internally in our organization. The .NET software characteristics: type and pointer safety, hard to make memory leaks are the main contributors to the quality. For non-expert C++ programmers, .NET languages (e.g. C#) appear to be a better choice.

It is important to mention that the extensible application developers were not able to ensure protection for faults like loops and locks, as well as memory leaks in the extension code, with the help from .NET. This has been achieved through an external supervision by a watchdog.

Figure 24 below illustrates a part of the product quality model of the device we have discussed. The left hand side of the figure shows a simplified product model of the GTPU device, with the design layers as discussed in Section 4.3. On the right hand side of the figure, a part of the quality model is shown. Links between component quality-carrying properties on the left hand side and some of the qualities are also shown. The model is partial in the sense that many properties are not mentioned both on the product model and the quality model side.

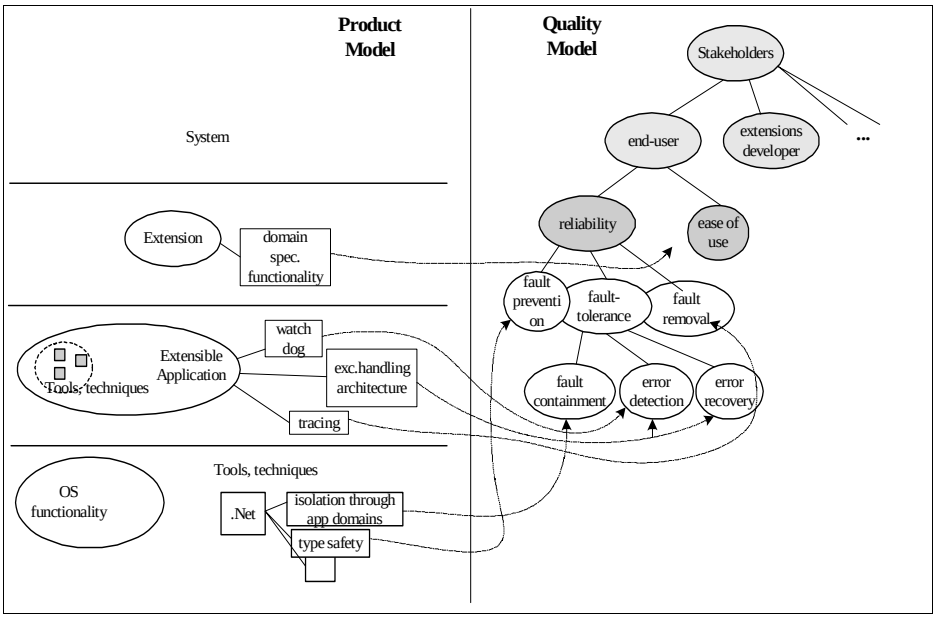


**Figure 24 Partial quality model of the FlexPendant device**

## 4.4 Conclusion

The problem of constructing a quality system is often presented in component-based software engineering (CBSE) as a problem of predictable assembly of components [21], where components are often treated equally in the terms of qualities they bring into the system. In systems that follow the layered architectural style, it makes sense to look at this problem using a layered approach, where OS is the most general and the lowest level component. Each of the layers brings certain design time and runtime qualities to the end system and also ensures that some qualities are present on the next level.

Figure 25 summarizes our reasoning on the effect of architecture on system dependability and openness. If we take a closed application with A1, and gradually publish internal interfaces making it open and extensible (from E1 towards E3), this will most likely decrease overall system dependability (from D1 to D3). However, it is possible to modify the system to architecture A2 (keeping other parameters constant), so that opening the system (from E1 to E2) keeps the desired level of dependability (D1) quality attributes.

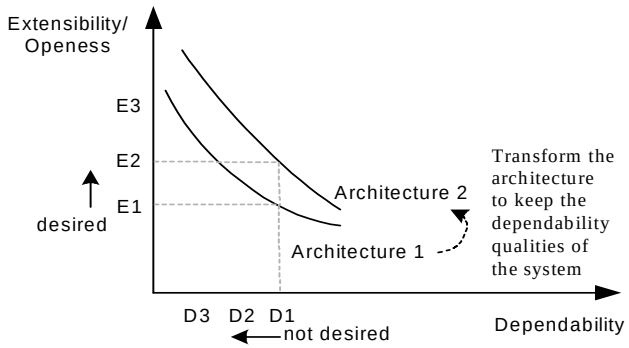


**Figure 25 Visualized tradeoffs between extensibility and dependability for two different system architectures**

In order to create a quality open architecture, we need to have in mind that there are two important user groups for the extensible application – the developers of extensions and the end-users. We need to have a clear picture of the relationship of the technologies we use and their contributions to the desired levels of quality, which is often hard to find. It is accumulated in experiences of the developers that use certain technologies. Because technologies today move very fast, it is hard to systematize such experiences and accumulate the knowledge that will be up to date.

# PART III – CONCLUSIONS AND FUTURE WORK

*"Architecting is characterized by dealing with ill-structured[4] situations, situations where neither goals nor means are known with much certainty.*

*An architect who needs complete and consistent requirements to begin work, though perhaps a brilliant builder, is not an architect."*

*Maier and Rechtin [32]*

This part of the thesis provides answers to the research questions stated in chapter Introduction, followed and by a critical discussion. The reasoning and conclusions in this part are based on the literature survey (brief overviews are presented in Appendix A) and the results of the case studies that are presented in Part I and Part II. Finally, we discuss opportunities for future work.

[4] Ill-structured problem: A problem where the statement of the problem depends on the statement of the solution. (Maier)

# 5 Conclusions

In this chapter we present answers to the research questions and hypotheses formulated in the introduction. We will also discuss the validations of the results. The two main research questions and hypotheses Q1/H1 and Q2/H2 will be discussed separately in the two subsections that follow. However, the research results presented in Part I and Part II are used in both subsections.

## 5.1 Question 1

***Q1:*** *What factors are considered important by software architects of complex embedded systems and have significant impact on software architecture design?*

***H1:*** *In addition to the general factors that are considered in architectural design of software systems, the factors that originate from relations between software, hardware and entire system must be taken into consideration.*

***Conclusion:*** *Based on the research results presented in Part I of the thesis, we argue that the hypothesis presented holds. For complex embedded systems, software architects work jointly with system and hardware architects in designing a solution, the solution being always the architecture of the whole system. Furthermore, the results and the conclusion from Q2 and H2 highlight some shortcomings of the architecture design methods, which have been developed for software systems, when they are applied in the context of complex embedded systems.*

### 5.1.1 Conclusion Details

In the conclusion of Part I, we have presented a list of findings that show certain common factors of concern for software architects in complex embedded systems. Those factors were presented within a product development life-cycle dimension, but many other alternative dimensions are possible. To support the hypothesis H1, and emphasize the factors resulting from the relationship of system, software and hardware to be considered, we now present the findings using the system-software relationship. The relationship is shown in Figure 26. The model is simplified because it does not show the fact that complex systems are hierarchical in their nature with subsystems containing both hardware and software. We have grouped the factors into groups A-D: *system level factors, relations between software, hardware and the entire system, architect's experience* and *development environment.*

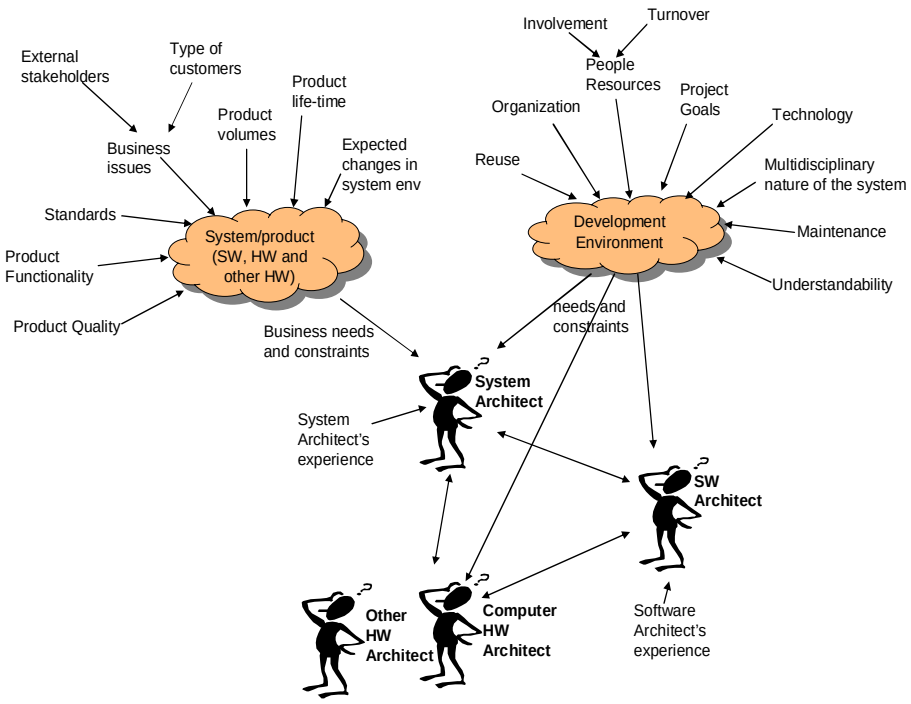


**Figure 26 A model of propagation of factors of concern in a complex embedded system. The fact that a system is hierarchical and that it is likely to have many subsystems which are hardware/software subsystems is not shown.**

A) *System level factors* that influence system architecture on the product level:

1. *Types of customers* - certain customers have much impact not only on what functionality a system will have, but also how it should be designed and implemented.
2. *Product operational quality attributes* - only the most important extra-functional requirements that are dominating for the specific type of systems are explicitly considered and taken into account in architectural design. Depending on the type of system, extra-functional hardware requirements may get more attention than software.
3. *Standards* - if it is feasible, an implementation of a standard is assigned to a subsystem. For systems with mature and standardized domains, more explicit attention tends to be given to extra-functional requirements. Additional features and distinguishing quality characteristics provide a basis for competition.

Isolating the implementation of standards in clearly defined subsystems favours system evolution as standards tend to change and evolve slowly.

4. *Other factors* – such as: product volumes, product lifetime, expected changes in the system environment are also of concern and were discussed in Section 2.4.

B) *Relationship of entire system, hardware and software*:

1. *The system is the primary concern and system thinking dominates the architecture design* - systems are decomposed into subsystems, these often including both hardware and software. The exact allocation of hardware and software requirements is performed on the subsystem level. The system architecture and the hardware resources place restrictions on the software, and in addition, the *software architecture* on subsystem level tends to be relatively independent of the system architecture. There are systems in which the hardware design is not complete when the software design and implementation is started. Since software is inherently more flexible, certain late problems (in development lifecycle terms) originating from hardware, are often solved by means of appropriate software design.
2. *Hardware or software dominance* - the overall importance of software is dependent on the type of systems that we have investigated:
   - Because software is used in an increasing number of functions, the importance of software is increasing and software architects are increasingly engaged in top level system design decisions, not only in software system design. As the efforts spent on software development increase, the investment in software is becoming proportionally greater than that of the hardware.
   - The "hardware first" approach is still dominant in many systems, as reported in our own and related studies [20]. A clear example is seen in telecommunication systems. Written comments to the presented paper, from a Nokia architect, support this statement (see Section 2.6).
3. *Allocation of requirements between subsystems* is often complex and not easily understood and for this reason must be explicitly addressed. This problem was observed during the case studies presented in Part II. Before the impact of opening up the system could be analysed, it was necessary to construct an analysis model.

C) Architect's *experience in complex embedded systems*:

1. *An architect needs to have the skills of both a generalist and a specialist* – an architect is a technically very competent person, able to handle both the detailed technical issues in implementation as well as the coarse-grained big picture.

2. *Knowledge of hardware and general knowledge of software is needed by a software architect* - experience from the study revealed that architects have quite a profound knowledge of hardware, even if software architects may not be completely aware of it, or consider it important.
3. *Architects need to have good knowledge of the system application domain*.

D) *Development environment* factors that influence the architecture:

1. *Reuse of subsystems is considered in architecture design* - experience was found to be more valuable than subsystems and code, but the reuse of subsystems (hardware and software) is still considered an important economic factor. As software is getting becoming more complex, reusing software components is becoming increasingly important.
2. *Making a choice of technology is an important architectural decision* - this was not at first obvious from the interviews, but crystallized after the analysis of interview data.
3. *A multidisciplinary nature of the system will require a multidisciplinary approach to detailed design and implementation* – this needs to be considered and is likely to have an impact on the partitioning of the system into subsystems and components.
4. *Organizational (development) perspective of architecture needs to be considered* – this is in order to minimize the interdependencies in the software that make integration and validation more difficult and also to achieve clear interfaces between different organizational units. *Packaging* of software modules is important and in some sense independent of the operational architecture, i.e. the packaging structure defines an architecture of its own that is tailored to the structure of the development organization/project. We have seen an example of this for an open system in Section 3.5.2.
5. *A balance needs to be found between long term strategic goals vs. short term project goals* - there is a frequent problem in balancing long-term strategic goals, which are often associated with the system's architecture as a valuable long term asset, and short term project goals. One way of avoiding this problem is to convince project leaders and managers of the importance of a long term investment in architecture. Alternatively, as suggested in [50], architects should not be subordinate to project managers responsible for individual development projects.
6. *The process view of architecture should not be neglected* – it is related to the organizational perspective of the architecture and if neglected, the testability of the product and intermediate product components may be impaired.
7. *Architectural design takes time* – it is not something which is performed in days or weeks. In the case of large systems, it may take years.

8. *Architectural design is the responsibility of a core team that consists of experienced designers* - the core team develops architectural principles (typically manifested as guidelines, handbooks) and the core infrastructure (platform). The base principles and the infrastructure should be relatively stable before a large scale development is started.

### 5.1.2 Validity of the Conclusions

It is the architects' *experiences* that are the main sources of the data on which our conclusions are based. As pointed out in the previous section, a critical analysis of the collected data reveals certain aspects of the discussed topics, which we believe are important but were not emphasized during the interviews. There is a difference between the *real* problems and the problems that the architects *think* are the real problems, and therefore, additional methods are necessary to find the problems of which the architects may remain unaware.

In certain cases, a direct answer may differ from the answer revealed after a more detailed analysis of the interview data. An example is the item "if a choice of technology is a part of architectural design". Others have reported similar experiences with contradictory survey data, and a recent example from requirements engineering research area is mentioned in [41].

Another possible problem involving the results obtained is that they give an echo of the problems that are hypothesized in literature. To avoid this problem and the problem of architects misinterpreting our questions during interviews, we used the questions as an outline of the topics to discuss and we tried to get as many examples as possible, which backup the architect's statements. We believe that the following factors give us solid backing for the validity of our conclusions: our analysis of the interviews data, a workshop with the architects, reviews of the published paper by four experts in the software architecture field and positive feedback at the conference.

Given that all the interview data has been collected in Sweden, a generalization to a wider context could be questioned. A similar study of embedded systems in Europe [20] has come to similar conclusions with respect to those topics of investigation that overlap, so we believe that the results of our study can be widened to relate to complex embedded systems in a European context. Based on the conference and review feedback, we believe that the conclusions regarding relationship of software, hardware and system are relevant for complex embedded systems and less applicable outside that context, while other conclusions are applicable to software architecting of complex systems in general.

### 5.1.3 Lessons Learned

*It is not easy to identify clearly what software architecture is*. The most common definition used in the literature is "structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them." [7]. If components and connectors are easily reconfigurable, then the value of getting them right is of less importance. A definition of what is architectural is then much more in favour of the

definition provided by Fowler[5] in [19]. Others also report that the difference between architecture and design in practice is far from obvious [20]. One of the definitions of architecture suggested by certain attendees at the WICSA'04 was "architecture as a structured set of design decisions." which we would like to extend to the following definition: "Software architecture is a specification of a structured set of design decisions that affect most of the system components and are difficult and/or expensive to change." It is recognized that the qualities of a product depend not only on its architecture but also on the implementation and quality of the system components. However, it is possible, at the level of the system architecture, to foresee that choices of e.g. certain technologies, programming languages will have a significant impact on the development environment, the number of faults introduced in the development, the performance of the system as built etc. Realizing late in the development life-cycle that incorrect decisions were made is very likely to have a significant negative impact on the system; it may be necessary to change its structure, the project may significantly miss its time schedule etc.

Another observation is that there is a difference between *architecture* and *architecting*. The Architecture of software is the result of the architecting process. This architecting process is much broader in scope than the architecture itself. It requires that architects work with requirements engineers and engineers who implement the system.

### 5.2 Question 2

***Q2**: Which dependability means should be applied[6] on the software architecture level to support system quality in an open system and what is a suitable approach to systematically applying those dependability means?*

***H2**: Quality attribute oriented software architecture design methods can be used as a systematic approach to applying dependability means in the context of open business-critical complex embedded systems.*

***Conclusion:** On the basis of our real-world experience from the development of open systems in industrial robotics, which are reported in Part II of this thesis, we argue that hypothesis H2 holds only partially. The principles of the state-of-art quality attribute-oriented software architecture design methods should be improved to permit better integration of software with system level architecture in complex embedded systems. This will support the traceability of quality attributes through system hierarchies and permit the performance of impact analysis on the system level.*

---

[5] "Architecture is the decisions that you wish you could get right early in a project, but that you are not necessarily more likely to get them right than any other" and architecture as "things that people perceive as hard to change".[19]

[6] As mentioned in the Introduction, we do not provide a hypothesis for the list of dependability means that need to be applied, but answer the question directly.

*The dependability means that need to be applied in software architecture design in order to support system quality are not limited to the application of fault-tolerance on the operational/runtime architecture of the system (such as implementation of fault containment zones). The dependability means should be implemented as a combination of fault-prevention, fault-tolerance and fault-removal techniques (a list is provided in Section 5.2.1).*

### 5.2.1 Conclusion Details

We first discuss the use of principles to systematically apply dependability means on the software architecture level for open systems (hypotheses H2). After that, we present a list of dependability means that were applied to support system quality of the open systems we have studied.

Dependability techniques are not limited to architecture design but need to be applied from requirements to verification and validation. In both case studies that are presented in Part II, we have shown that for an open system, development tool support should be decided during architectural design process. Modifications to the system's runtime architecture should be coordinated with the development architecture and level of confidence in fault prevention techniques that are supported by the tools. This leads us to the tradeoffs between dependability and development quality attributes. An example of a tradeoff that should be considered by an architect is usability vs. dependability; if we decide to use certain implementation technologies and tools to raise the level of confidence in fault prevention, the tools should be easy to learn, easy to use, not too expensive etc. Another example is the tradeoff between dependability and the reuse of existing components and architecture; it is desirable to minimize changes to preserve an investment in existing assets, but that is in conflict with the desired level of dependability and therefore results in changes of both operational architecture and support in the development environment. A method for systematically applying dependability means should therefore provide support in dealing with such tradeoffs.

Dependability is not only a quality attribute; it is a concept for the development of systems that can deliver services that can justifiably be trusted and as such, it covers a much longer part of the development lifecycle, compared with the methods for architectural design in general. On the other hand, as compared with dependability, software architecture design methods cover a wider range of factors that influence software architecture design by enabling systematic analysis of different quality attributes that the system needs to exhibit. The current disadvantage of software architecture methods based on quality attributes is that they lack good traceability between the properties of the whole system and the properties of the software. The properties of the entire system are of primary interest in complex embedded systems. In Chapter 3, we have shown that we needed to propagate system requirements to the controller subsystem software, and that was done on the basis of the top level system architectural decisions. The requirements on individual subsystems are allocated after system design and after dependability techniques

are applied at the system level. System level decisions and allocation of functionality are performed for both hardware and software subsystems. If we change a system property, such as system openness, this does not mean that we only change the same property, the openness, of the software subsystem. In fact we may not need to change this property for some of the software subsystems at all. Only after an impact analysis is performed on the system level, after decisions are made with respect to the distribution of functional and non-functional properties between system components, and decisions are made with respect to the distribution of requirements between hardware and software, can we know what changes should be made in the software of a subsystem. Efforts to establish a better link between the "system level" and "software level" in quality attribute based software architecture methods are a subject of current research, as mentioned in [43].

On the software subsystem level, the principles of quality attribute centric software architecture methods provide assistance in systematic design based on quality attributes and in dealing with tradeoffs. Dependability techniques cannot be applied in isolation from other techniques that are used to support other important quality attributes, such as non-operational quality attributes (e.g. reuse, business qualities) as discussed in Sections: 4.3, 3.5.2 and 3.5.3. Reuse, which is a factor of great importance in system architecture (as discussed in Section 2.4.2), requires, in system design, that top down analysis from the system level be combined with bottom up analysis, as seen in Chapter 4.

With respect to the inputs that are suggested by the quality attribute based software architecture design methods (SEI – Appendix A3.2 , Bosch – Appendix A3.1), we have the following comment: Based on our experience, we argue that developing one or more analysis models that include quality attribute requirements (such as described in Section 3.5) can be beneficial and used in addition to "a list of profiles" as suggested in [12] or *utility tree* suggested in [7]. Furthermore, when we deal with reuse during the design process, we often need to evaluate the impact of using one component vs. another, i.e. perform impact analysis on several quality attributes in parallel. In that case, our experience is that models such as *utility tree* or *NFR Framework* (Appendix A3.3) provide more support for impact analysis than *a list of profiles* [12].

Our second goal was to answer the question "which dependability means need to be applied on the software architecture level to support the quality of open systems". We group these dependability means according to the generally accepted dependability taxonomy (Appendix A.2) and present them below.

*Fault prevention* dependability means:

- Providing automated tools for open software system extension development and system integration, which are easy to use, is one of the primary objectives. We support this statement by means of case studies presented in Chapters 3 and 4.

- The choice of underlying technologies on which an open system is based has indirectly a significant impact on a number of faults introduced in system implementation (e.g. choosing between Win32 and .NET as seen in our case study, in Chapter 4).
- Interfaces of an open platform need to be clear, simple and unambiguously defined. Interfaces which do not have these properties will result in an increased likelihood of faults being present in the final system. Furthermore, there is a need to decide on which interfaces and variability points in the system are "public", i.e. available to the third parties. Interface qualities have a very significant impact on ease-of-use. It is highly unlikely that small organizations with few resources will be capable or willing to spend their resources on extending an open system that is difficult to use.
- In an open system, there is more need for the use of special tools for reasoning about properties such as timeliness. Qualitative reasoning about such properties is difficult and requires detailed knowledge of the system internals. As shown in our case-study (Section 3.6.3), we have therefore introduced a software tool for a quantitative evaluation of the system design that checks for timing related errors such as the violation of deadlines and queue underflow or overflow.

*Fault-tolerance* dependability means:

- Fault containment zones should be used to isolate critical functionality from non-critical functionality, as shown in the both case studies presented in Part II.
- Watch-dog techniques should be applied for monitoring system resources.

*Fault-removal* dependability means:

- The need for tool support for verification and validation was discussed in Section 3.6.3 (ART Framework tool for regression testing).
- Fault diagnosis that records the cause of errors in terms of both location and type is needed in order to be able to find the faulty component and correct the fault. Monitoring and debugging in the distributed real time system is challenging and the approach we have used to address it is mentioned in Section 3.6.2.

### 5.2.2 Validity of the Conclusions

The conclusions about the applied dependability means are based on our two case studies which are based on two development projects of two different subsystems within the same system. We believe that our general observations related to dependability means and software architecture design are applicable to other systems, where openness is of interest. Our conclusions can be an initial input for creating a catalogue of *tactics* (SEI terminology[7]) or *operationalizations* (NFR

[7] See Appendix A3.2.

Framework terminology[8]) when analyzing and designing a system where openness is of interest. However, in order to package our conclusions in a more systematic way, such as the "tactics", we would need to perform additional case studies based on other systems and further systematize the knowledge gained.

The type of the system, into which openness should be introduced, may strongly influence the choice of details of the applied dependability means. The system on which our case studies are based is a business/safety critical industrial robotics system. There are other types of systems to which our conclusions would be much less applicable, either because the required level of dependability confidence may be much higher (e.g. mission critical systems) or the priorities allocated to quality attributes to be achieved are different (e.g. a difference between a telecom switch and a robot controller).

We have also made certain observations about the state of the art of quality attribute software architecture methods. We state that they need further improvement and integration with system architecting methods in order to be more useful for designing complex embedded systems. We believe that these conclusions can be widened to the application of these methods to the design of other types of complex embedded systems, as they are more general in their nature.

### 5.2.3 Lessons learned

In [12], Bosch discusses the following three uses of software architecture:

- software architecture for a *single system*,
- software architecture for *product lines* and
- software architecture for *component framework*.

Based on our case study of the open robot controller, we believe that, if a system based on product line architecture is to become open, the product line infrastructure (the platform) becomes more like the *component framework*. We see the following difference between the *open product line architecture* and the component framework. In an open product line the market for third party components (custom-made for the platform) need not be explicitly addressed or it may not be of special interest (e.g. too small a market).

[8] See Appendix A3.3.

# 6 Future Work

The following are issues identified in Part I that we believe deserve further attention:

- Systems are constantly exposed to an incoming flow of new functional and non-functional requirements, new constraints etc. How should a design rationale be formulated and enforced in system evolution in general to prevent architectural degradation? In the majority of the cases in our study, tools or processes for handling new requirements were not used. As a consequence, the focus during system evolution was on the new requirements only. In such a situation, there is the risk that old requirements may be violated while new requirements are being implemented.
- No or limited strategies for communicating the important architectural principles were found in the case studies. Plain language is currently the predominant way of documenting system and subsystem requirements, using the word processor for specifications based on company standard templates. We found no clear strategies for propagating requirements and design decisions, especially for non-functional requirements. This supports the statements previously made by others [20]. Because of the complexity of the systems, our experience from industry is that even the most experienced engineers may have an insufficient understanding of the allocation of extra-functional requirements to subsystems, even for a mature system.
- When the product lifetime is 20-30 years, any unavailability of software components, for example, the OS used to build the original system, is as much a problem as the unavailability of the hardware used in the original system design. We have mentioned an example of project in which it was decided to use the Linux OS platform instead of MS Windows, because it was believed to be preferable to own the source code.

Based on our research results in Part II of this thesis and the current state of the art of software architecture design methods, we believe that there are several issues that deserve further consideration as a subject of future work. More work needs to be done to improve the integrated modelling of system and software properties through the hierarchy of systems and subsystems. SEI is currently investigating the applicability of their methods in software intensive systems and scenarios, tailoring them to specific organization needs and investigating integration with RUP and RUP SE (see Appendix 3.2). We believe that tailoring methods are not only needed for different organizations, but also for different groups within organizations (subsystems within a system). We believe therefore that more work could be done on crystallizing the basic principles that are common to methods such as: NFR Framework, ATAM, Bosch's method etc. These basic principles could then be adapted by development teams to their own requirements. This is an approach different from that of SEI, which introduces SEI methods "organically" into the development processes used by other organizations [28].

# APPENDIX A - TERMINOLOGIES AND OVERVIEWS OF THE RESEARCH AREAS RELEVANT IN THE CONTEXT OF THE THESIS

In Figure 2 in the Introduction, the following research areas were identified as relevant to the research question of interest in this thesis:

- Software Quality
- Dependability
- Software Architecture

In this section, we provide a short overview of the terminology and the most important concepts from these areas, which are specifically relevant in the context of this thesis.

## A.1 Software Quality, Quality Models and Quality Attributes

Many researchers have studied quality and one frequently seen and cited view of quality is the one given by David Garvin's in 1984, which says that "quality is a complex and multifaceted concept", which he describes from five different perspectives. These perspectives were discussed in the context of software engineering by Pfleeger and Kitchenham in [46]:

- *Transcendental view* - In this view, software quality is thought of as an ideal towards which we strive, but it can never completely be reached; it can not be defined but you know what it is.
- *User view* - In this view, software product quality is seen as fitness for purpose/use, evaluated in a task context, and can thus be highly personalized.
- *Manufacturing view* - sees quality as conformance to a specification and focuses on the product quality during production.
- *Product view* - Advocates of software metrics often use the product view approach, which looks inside of a product as opposed to user and manufacturing views, which are views from the outside. The advocates assume that measuring and controlling product internal /inherent properties will result in better external product behaviour. Certain models link the product view to the user view.
- *Value-based view* - In this view, quality is dependent on how much a customer is willing to pay for it. In the real world, requirements on a product often change during its development, and trade-offs between cost and desired quality have to be made. The value based view can be helpful to resolve the issue. In this context, internal software measures are irrelevant.

The perspective we take on the quality influences the way we define it but also the way we measure it. The "users view" is often used in software engineering and users are often referred to as stakeholders. Measuring quality in the user's view is problematic because desired properties on the high-level (such as e.g. a need to have a safe, reliable, usable product) are intangible. Generalizing Tom Gilb's techniques, Kitchenham and Pfleeger state that "the quality concept is broken down into component parts until each can be stated in terms of directly measurable attributes". Decomposing quality into various factors leads us to various quality factor hierarchies, commonly referred to as *quality models*, such as: McCall's quality model, Dromey's [17] model, Bohem's model [9] [10], ISO 9126 [23], IEEE 1061-1998 [1]. These models use different terminology for the different nodes and levels of their hierarchies; *quality attributes* being one of them. Furthermore, a difference can be made between fixed models and flexible models.

**IEEE 1061-1998** defines software quality as *the degree to which software possesses a desired combination of quality attributes*. *Quality attribute* is defined as characteristic of

software, or a generic term applying to quality factors, quality subfactors or metric values (see Figure 27).

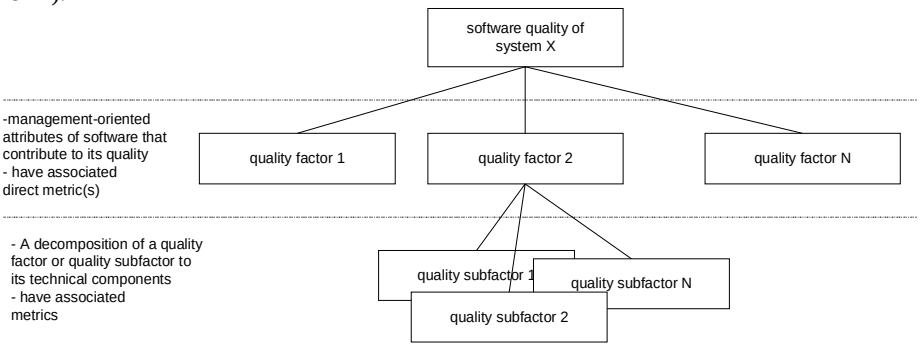


**Figure 27 IEEE Std 1061-1998 software quality metrics framework**

Metrics or software quality metric is defined as "a function whose inputs are software data and whose output is a single numerical value that can be interpreted as the degree to which software possesses a given attribute that affects its quality." Direct metrics is the metric that does not depend upon a measure of any other attribute.

While IEEE Std 1061-1998 does not provide a list of factors, earlier version of this standard, IEEE Std 1061-1992, lists the following factors and sub-factors as "sample" in annex A part of the document: *Efficiency* (time economy, resource economy), *Functionality* (completeness, correctness, security, compatibility, interoperability), *Maintainability* (correctability, expandability, testability), *Portability* (hardware independence, software independence, instalability, reusability), *Reliability* (non-deficiency, error tolerance, availability), *Usability* (understandability, ease of learning, operability, communicativeness).

In **ISO 9126-1**, quality is defined as the totality of characteristics of an entity that bear on its ability to satisfy stated and implied needs. A quality model is defined as the set of characteristics and the relationships between them which provide the basis for specifying quality requirements and evaluating quality. In the quality model, the following terminology is used: software characteristic, subcharacteristic and *quality attribute*. *Attribute* is a measurable physical or abstract property of an entity. ISO 9126-1 quality model framework is shown in Figure 28.

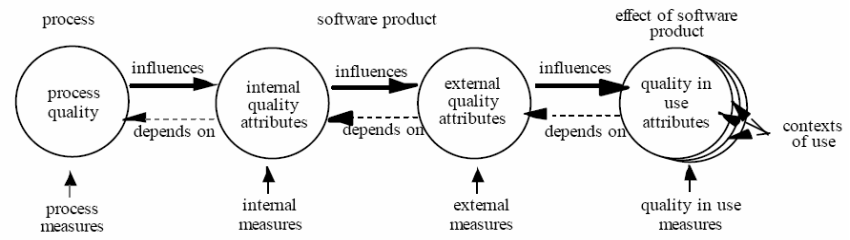


**Figure 28 ISO 9216-1 model of quality in the software development lifecycle [23]**

ISO 9126 defines the following attributes in the quality model for *quality in use*: effectiveness, productivity, safety and satisfaction. The following quality attributes are common for the *external* and *internal* quality model: functionality (suitability, accuracy, interoperability, security, functionality, compliance), reliability (maturity, fault tolerance, recoverability, reliability compliance), usability (understandability, learnability, operability, attractiveness, usability, compliance), efficiency (time behaviour, resource utilization, efficiency compliance), maintainability (analyzability, changeability, stability, testability, maintainability compliance), portability (adaptability, installability, co-existence, replaceability, portability compliance).

In software engineering, properties related to software-intensive systems are usually classified into functional and non-functional ones [48]. Preiss points out in his PhD thesis [48] that all the properties exceeding the system's main input/behaviour (it's main services for the direct users), are inconsistently called one of the following: *Ilities*, *non-functional properties* (in most literature) , *afunctional qualities*, *extra-functional properties* or simply *quality attributes*. Some authors distinguish between these terms and while many use them almost synonymously. Based on *theory of properties and systems* that is described in the thesis and the current use of the terms in software engineering, Preiss proposes the following core terminology:

- *Attribute/property* – as defined by standard dictionaries
- A *required attribute/property* – a need or desire on an entity by some stakeholder. Also called *requirement*. Because there are different types of properties, there are different types of requirements (e.g. functional, extra-functional).
- An *exhibited attribute/property* – a property ascribed to an entity as a result of evaluating (either directly or indirectly) the entity.
- *Quality* – the totality of exhibited attributes/properties of an entity that bear on its ability to satisfy stated or implied needs (i.e. have relationship to required properties).
- *Quality Attribute/property* – refers to an exhibited attribute/property that is a part of *Quality* of an entity.

- *Functional property* – used to denote the primary functions of a component/system relative to its operational context and hierarchical level. They can be observed as a response to stimuli in the system's operational context.
- *Non-functional properties* – all but functional properties of a system.
- *Afunctional (developmental) properties* – the properties of a component/system seen in its development context.

Furthemore, Preiss has defined a flexible quality model and a quality model construction framework. Fixed quality model are found insufficient to describe all possible properties which are invented by humans and serve to explain phenomena of interest.

Preiss also discusses three different but related types of system property decompositions: *classification oriented* (its primary use is structuring knowledge), *analysis-oriented* (its main purpose is trade-off and impact analysis) and *realization oriented* (used to relate system-level properties to the elements that realize the system).

## A2. Dependability

*Dependability of a computing system is the ability to deliver service that can justifiably be trusted* [5]. To clarify this definition, we need to define what a *service* is. Functionality of a system is provided to users of the system as a service and through an interface called the *service interface*. *Users* are considered as another system. System may fail because it does not comply with the specification, but also because the specification does not adequately describe its function. Dependability can be associated to a set of attributes, means and impairments (Avižienis et al. [5]). The taxonomy is presented in Figure 29 below.

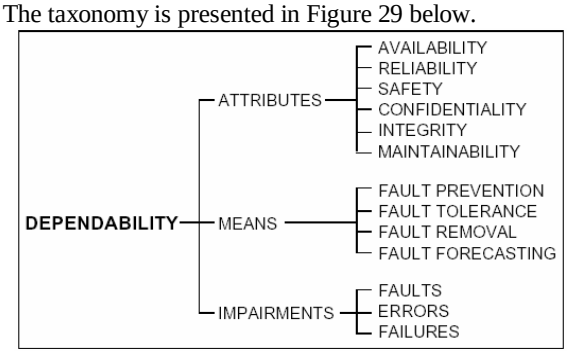


**Figure 29. The dependability tree. "Threats" to the "attributes" are achieved by the "means" [5]**

The definitions of dependability attributes are based on Dependability Handbook [30]:

- Availability is readiness for usage
- Reliability is service continuity
- Safety is non occurrence of catastrophic consequences
- Confidentiality is non-occurrence of inadequate information disclosure
- Integrity is non-occurrence of inadequate information alteration
- Maintainability is an ability to perform corrective and evolutionary updates to software.

Note that *security* is a composite attribute, which combines confidentiality and integrity. *Robustness* is defined as a secondary attribute; it is dependability with respect to external faults [5].

One of the strengths of the Dependability concept is the unified view of its threats – faults, errors and failures.

- *Failure* is the event of system transition from correct to incorrect service. The opposite event is called "service restoration" and the time in between is called "outage time".

- *Error* is the part of the system state that causes the failure. Not all errors become failures. If the error propagates to the service interface boundary and alters the service, it becomes a failure. Errors can be detected and reported by some means of error log/message. Undetected but present errors are called *latent* errors.
- *Fault* is the adjudged or hypothesized cause of the error. Faults can be *active* (producing errors) or *dormant* (not producing errors). There are many fault classes and the classification may vary depending on the view chosen.

After defining faults, error and failures, we can also define some terms related to the behavior of systems in case of errors are detected, e.g. classify them according to the way they can fail.

- *Fail-controlled* are systems that are designed and implemented so that they, to an acceptable extent, fail only in specific modes of failure described in the dependability requirements.
- *Fail-safe* systems are a type of fail-controlled systems that, to an acceptable extent, fail in such a way that can cause no harm to the environment.
- *Fail-halt (or fail-stop)* are also a type of fail-controlled systems, which go to a predefined state upon detection of an error. According to the output produced upon transition to this predefined state, they can be divided to:
    - *Fail-silent* – no output is produced
    - *Fail-passive* – a predefined output is produced

A unified view of fault, error and failure opens a possibility to better structure different means for fighting those threats. Means for dependability are "the methods and techniques giving the system the ability to deliver a service conforming to the accomplishment of its function and to place a trust in this ability [30]." There are four different means for dependability:

- *Fault Prevention* - Includes all activities like: programming methods, processes, formal methods etc, which can help to avoid making errors.
- *Fault-Tolerance* - Includes all techniques which are used to prevent runtime errors to turn into faults, or in other words to prevent system service from failing. These include: error detection, error and exception handling, fault handling (isolation of detected faults in runtime).
- *Fault Removal* - Includes all development time activities that remove faults in the design and implementation (through: verification, diagnosis and correction) and the specification (validation). The most common ways of verification are: testing, code reviews and static analysis.

- *Fault Forecasting* - Includes all techniques used to predict the amount of remaining undetected faults and also predict the future system behavior.

A detailed analysis and fine detailed classifications of the above means (except of the fault prevention) can be found in the Dependability Handbook [30] and a shorter overview of the measures can be found in [5].

To systematically apply dependability means, Kaaniche, Laprie et al propose *dependability-explicit development model* [25]. This model includes three main processes, which are parallel to each other in time:

- System creation process
- Dependability process (Fault Prevention, Forecasting, Tolerance and Removal)
- Certification and QA process (quality assurance and certification process)

Requirement, design, realization and integration are usual steps in a development process and the dependability-explicit model implies no temporal sequencing of these activities, because it is not a classical life-cycle model, but can be viewed more as a *meta-model*. The Dependability process includes four processes and each of the processes corresponds to one of the means for dependability. More details can be found in Dependability Handbook [30] and the paper [25].

The concept of Dependability is rather similar to the concepts of Trustworthiness and Survivability. The summary of differences is presented in [5]. In some sources (e.g. Sommerville [56]) terms dependability and trustworthiness are treated synonymously.

To conclude, we can say that dependability is not just a quality attribute but an approach to development of critical systems in which dependability attributes are the primary concern.

## A3. Software Architecture

In this section we describe several approaches to software architecture design and evaluation based on quality attributes.

### A3.1 Bosch's approach

In [12], Bosch discusses three ways to use software architectures: a) for an individual system, b) as product line architecture and c) as a standard architecture used for a component market. He presents a software architectural design method for the case a) individual systems, and complements to this method to make it suitable for the case b) product line architectures. The basic method is called *quality attribute oriented software architecture design method* (QASAR) and consists of two iterative loops, which are depicted in Figure 30.

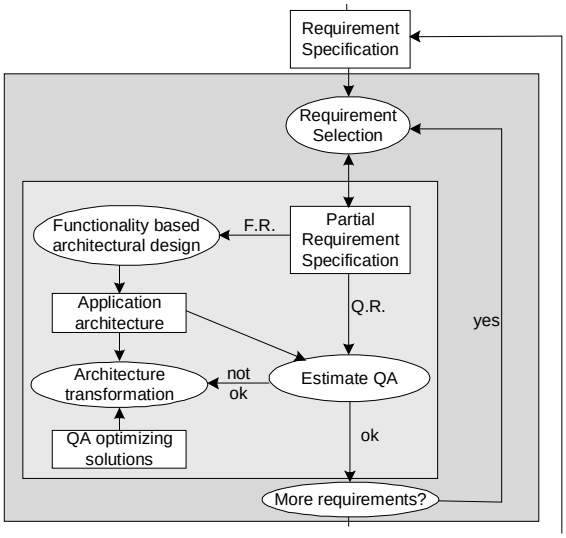


**Figure 30 Quality Attribute based transformations in Bosch's method [12]**

This method is iterative, in the sense that the inner loop is started with the core subset of the requirements. Based on them a first version of the architecture is developed. The focus of the inner loop is assessment of and transformation of the architecture for *quality attributes*. As opposite to the SEI methods, this methods starts from functionality-based architectural design, followed by estimation of quality attributes and finally transformations of architecture in order to fulfil quality attributes. An initial functional architecture will still implicitly have certain values for its quality attributes, but at this point they are not explicitly evaluated. This method defines the design process and the key artefacts that are developed during the process.

Requirements are divided into *functional* and *quality requirements* (the name *quality attributes* is used as a synonym to quality requirements). Quality requirements are categorized as operational or development requirements and have associated profiles, such as: usage profile, hazard scenarios and change scenarios. No specific recommendation is given for how to group or organize the quality attributes in relationship to each other. Techniques that can be used for quality attribute evaluations are: scenario-based evaluation, simulation, mathematical modeling and experience-based assessment. The following types of transformations based on the quality attributes are proposed:

- impose architectural style
- impose architectural pattern
- apply design pattern
- convert quality requirements to functionality
- distribute requirements

As previously mentioned, the basic method can not be applied directly to product lines. This is because it does not take required variability and differences between the products sufficiently into account. An extended method that takes these issues is also presented in [12].

The main characteristics of this method as described in [12] are still unchanged, compared to the more recent material presented by Bosch in tutorial on WICSA'04 conference [11].

### A3.2 SEI Architecture-Centric Methods

The Software Engineering Institute (SEI) at Carnegie-Mellon University, Pittsburgh, PA USA has developed a set of methods for software architecture-centric development. In the SEI approach, a set of influences depicted in Figure 31 via architect determine the architecture. This model of architectural influences is called ABC (Architecture Business Cycle).

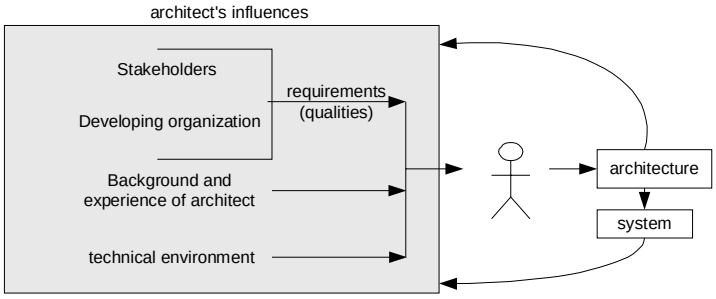


**Figure 31 ABC - Architecture Business Cycle [7]**

Architecture is shaped by some combination of: functional requirements, business requirements and quality requirements. The most dominant of those requirements are called

architectural drivers. Functionality is defined as the ability of the system to do the work for which it was intended and it is considered to be largely independent of the system's structure. In the SEI approach, no precise definition of quality attributes is given, but the term is used in such a way that it covers all but the functional properties of the system (i.e. it is used as a synonym to non-functional requirements). The following three *classes of quality attributes* are used in [7]:

- *Qualities of the system*, which apply directly to the system, such as: availability, reliability, modifiability, performance, security, testability, usability.
- *Business qualities*, which apply to the business environment, such as: time to market, cost and benefit, projected lifetime of the system and targeted market.
- *Qualities about the architecture itself*, which indirectly affect other qualities, such as conceptual integrity, correctness and completeness and build-ability.

Essential concepts for recording quality attribute specific requirements are scenarios. Scenarios include the following elements presented in Figure 32: the stimulus of the scenario, the source of the stimulus, the response, the response measure, the artifact stimulated and the environment.

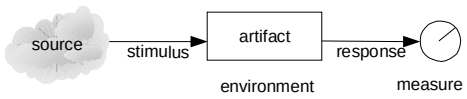


**Figure 32 Elements of a quality attribute scenario**

Scenarios are elicited from stakeholders and each scenario may have impact on multiple quality attributes. They are used in the *design* step and *analysis* step of architecture design process. The following types of scenarios are used [26]: use cases, growth scenarios (cover anticipated changes to the system) and exploratory scenarios (cover extreme changes that are expected to stress the system).

The architecture-centric methods are developed to support the architect in the ABC process and those methods complement each other in providing the support. The common set of characteristics for these architecture centric methods is [28]:

1. They are scenario driven (scenarios serve as an "engine" for directing and focusing the method's activities)
2. They are directed by operationalized quality attribute models
3. The methods focus on documenting the rationale behind the decisions made
4. All of the methods involve stakeholders so that multiple views of quality can be expressed.
5. Methods were designed as standalone and to be performed by an outsider, not as an integrated part of a software development cycle.

The following architecture-centric methods are developed:

- *QAW (Quality Attributes Workshop)* - The aim of QAW is elicitation and documentation of quality attributes requirements, where requirements are elicited and documented as six-part scenarios.
- *ADD (Architecture-Driven Design method)* - It is a method for defining software architecture by basing the design process on the system's quality requirements.
- *ATAM* (*Architecture Tradeoff Analysis Model)* - ATAM is a successor of the method called *SAAM - Software Architecture Analysis Method*). It is an evaluation method that reveals how well an architecture satisfies a particular goals and it provides an insight into how quality goals interact; how they tradeoff [7]. As a part of the evaluation, a hierarchical model of quality attributes is created which is called *utility tree*. Leaves of the utility tree are quality scenarios. Quality attribute names are used as an organizing vehicle, but the structure of the model is flexible and it allows for names that are the most suitable to the participants of the evaluation process.
- *CBAM (Cost Benefit Analysis Model)* - As an evaluation ATAM is missing important considerations – those that have to do with economic. CBAM is a method of economic modeling of software systems, centered on an analysis of their architecture [7].
- *ARID (Active Reviews for Intermediate Design)* - ARID is a method that blends Active Design Reviews [ref] with ATAM and helps to find issues and problems the successful use of the design. It is intended for use with partially complete designs.

One of the common characteristics of the methods, that they were designed as standalone and to be performed by an outsider is one of their drawbacks and it has recently been recognized by the designers of the methods [28] [27]. Organizations that wish to include SEI Architecture-Centric methods in their development life-cycle need to synchronize the activities of the methods with other development activities. If multiple methods are used, activities needed by the methods need to be synchronized to avoid duplication of effort. Addressing these problems is subject is the subject of the most recent and current efforts at SEI. The following technical reports provide more information on this subject:

- Lifecycle view of Architecture Analysis and Design Methods [28]
- Integrating ATAM and CBAM [42]
- Integrating Software-Architecture-Centric Methods and RUP [27]
- Integrating QAW and ADD [43]

Of special interest for this thesis is integration of QAW and ADD, where it is recognized that many of the scenarios are system oriented and that they need to be transformed before they can be used directly in the ADD method, which is applicable for software [43]. Integration to RUP and RUP SE is also of interest in the context of complex embedded systems.

### A3.3 NFR Framework

NFR Framework is described by Chung and colleagues in [14]. It is a disciplined approach to representing and analyzing non-functional requirements. In this approach, requirements are divided to functional (what the system does) and others – non-functional requirements (NFRs). Quality attributes is considered as a synonym to non-functional requirements. The authors clearly distinguish two different approaches for systematic treatment of non-functional requirements – *product oriented* and *process oriented* and in an orthogonal dimension – *qualitative* and *quantitative.* The product oriented approach is based on evaluation of software for how well it meets its non-functional requirements. NFR Framework approach is process oriented and provides techniques for justifying design decisions *during* the software development process. It is also qualitative in its nature because of the problem to quantitatively measuring an incomplete software system. Authors state that NFRs are difficult to deal with because they are: *subjective* (different people), *relative* (their interpretation depends on the system) and *interacting* (achieving one NFR may hurt one or more other NFR-s).

NFR Framework organizes non-functional requirements in a hierarchical structure of *softgoals* which can be *satisficed* rather than satisfied[9]. The structure is called *SIGs – softgoal interdependency graphs* and records developer's consideration of softgoals, and shows interdependencies among softgoals. The framework has well defined graphical notation for all elements that are included in the SIGs graphs.

Another important part of the framework is the knowledge catalogues. There are three kinds of catalogues: *type catalogues* (provide terminology and classification of NFR concepts), *method catalogues* (development techniques for achieving NFRs) and *correlation catalogues* (interdependencies between NFRs). The hierarchy of NFRs in the SIGs is not fixed. NFRs can be decomposed either based on their type or topics. Intangible softgoals are linked to operationalizing softgoals (the development techniques) that can be implemented. An example of a SIGs graph is shown in Figure 33.

[9] Softgoal have no clear-cut definition and/or criteria as to whether it is satisfied or not. Therefore the authors use the terminology satisficed rather than satisfied for the softgoals.

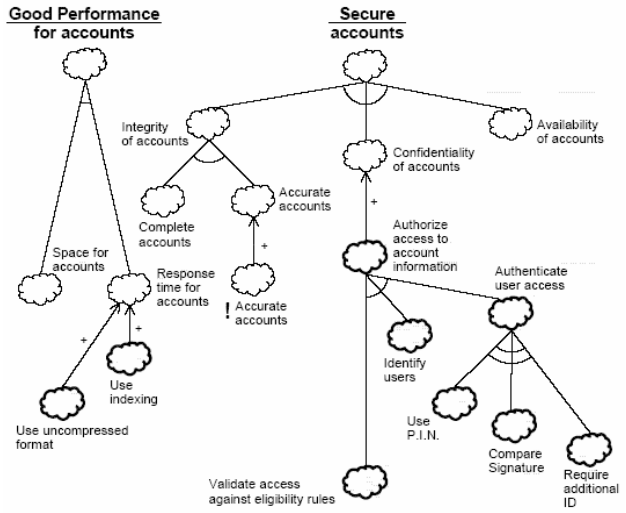


**Figure 33 An example of Softgoal Interdependency Graph in NFR Framework [14]**

NFR Framework can be applied to software architecture design to systematically guide a software architect in selecting among architectural alternatives. An example is provided in Chapter 12 in [14].

### A3.4 Other approaches

There are many other methods of architecture design that are based on the quality attributes or non-functional requirements. Several such methods are mentioned in this section.

A framework named NFD – Non-functional decomposition was developed by Poort [47]. In this approach, requirements are divided as depicted in Figure 34.

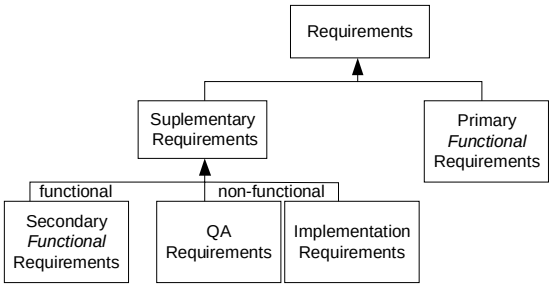


**Figure 34 Requirements classification in NFD method [47]**

*Primary Functional Requirements* are defined as demands that require functions which directly contribute to the goal of the system, or yield direct value to its users. *Secondary Requirements* are all other requirements imposed on the system. *Secondary Functional Requirements* require functionality that is secondary to the goal of the system. *Quality Attribute Requirements* are quantifiable requirements about the system's (operational) quality attributes. *Implementation Requirements* are those that can not be measured by system assessment. In this approach, importance of linking system architecture and development process is emphasized.

Many other examples of methods related to architecture and requirements can be found in the proceedings of the workshops STRAW '01 [2], and STRAW'03 [3] which were held in the conjunction with ICSE conference. For example, Kazman and Olson present in [22] an integrated decision-making framework from software requirements negotiation to architecture evaluation based on WinWin [4] and CBAM [7].

### A3.5 Certain common principles of the quality attribute centric software architecture design methods

The classification of the requirements and the terminology used in the methods that we have surveyed, supports Preiss statement that we have not yet reached consensus on terminology, classification and organization of system requirements and properties. This makes it hard to find the commonalities, i.e. the common principles used by different software architecture design methods and to define what a quality attribute based software architecture design method is. Still, based on the survey presented in this Appendix, we find that the current quality attribute based software architecture design methods have the following common characteristics:

- The term quality attributes is used as a synonym to non-functional requirements.
- It is required that quality attributes are explicitly elicited and recorded (e.g. through quality scenarios).
- Quality attributes are recognized as the main drivers that shape the architecture of software.
- Methods describe how to evaluate software architectures for fulfilment of quality attributes and how to transform the architectures to fulfil the quality attributes.
- Guidance is provided for dealing with tradeoffs between quality attributes.

# REFERENCES


[1] *IEEE Standard for a Software Quality Metrics Methodology*, IEEE The Institute of Electrical and Electronics Engineers, Inc., 1998, ISBN 0-7381-1059-6.

[2] http://www.cin.ufpe.br/~straw01/, STRAW'01 First International SofTware Requirements to Architectures Workshop, 2001.

[3] http://se.uwaterloo.ca/~straw03/ProceedingsSTRAW03.pdf, STRAW'03 Second International SofTware Requirements to Architectures Workshop, 2003.

[4] *WinWin homepage at USC/CSE*, http://sunset.usc.edu/research/WINWIN/, 2004.

[5] Avizienis A., Laprie J.C., and Randell B., Fundamental Concepts of Dependability, report 1145, LAAS, 2001.

[6] Baragry J. and Reed K., "Why we need a different view of software architecture", Working IEEE/IFIP Conference on Software Architecture, 2001.

[7] Bass L., Clements P., and Kazman R., *Software Architecture in Practice*, Addison-Wesley, 2003.

[8] Blom M., "Semantic Aspects in Software Development", ISSN 1403-8099, Karlstad University, Sweden, 2002

[9] Boehm B., *Characteristics of Software Quality*, North-Holland Pub. Co., 1978.

[10] Boehm B. and In H., Identifying quality-requirement conflicts, *IEEE Software*, volume 13, issue 2, 1996.

[11] Bosch J., Architecture-centric Software Engineering, 4th Working IEEE/IFIP Conference on Software Architecture, Tutorial TA1, Oslo, Norway, 2004.

[12] Bosch J., *Design & Use of Software Architectures*, Addison-Wesley, 2000.

[13] Bril R.J., Postma A., and Krikhaar R.L., "Embedding Architectural Support in Industry", International Conference on Software Maintenance, 2003.

[14] Chung L., Nixon B., Yu E., and Mylopoulos J., *Non-Functional Requirements in Software Engineering*, Kluwer, 2000.

[15] Conway M., *How do committees Invent?*, Datamation, 14 (4), 1968.

[16] Dromey G., A Model for Software Quality, *IEEE Transactions On Software Engineering*, volume 21, issue 2, 1995.

[17] Dromey G., Cornering the Chimera, *IEEE Software*, volume 13, issue 1, 1996.

[18] Dromey G., http://www.sqi.gu.edu.au/publications/SPQ-Theory.ps, Software Quality Institute, Griffith University, Australia, 1998, Software Product Quality: Theory, Model and Practice.

[19] Fowler M., Who needs an architect?, *IEEE Software*, 2003.

[20] Graaf B., Lormans M., and Toetenel H., Embedded Software Engineering: The State of the Practice, *IEEE Software*, 2003.

[21] Hissam S., Stafford J., Wallnau K., and Moreno G., "Packaging Predictable Assembly", Proceedings of the First IFIP/ACM Working Conference on Component Deployment, Berlin, Germany, 2002.

[22] In H., Kazman R., and Olson D., "From Requirements Negotiation to Software Architectural Decisions", STRAW'01 International SofTware Requirements to Architecture Workshop, 2001, 2001.

[23] ISO/IEC, International Standard ISO 9126-1, ISO/IEC, 2001.

[24] Issarny V., "Software Architectures of Dependable Systems: From Closed To Open Systems", ICSE 2002 Workshop on Architecting Dependable Systems, 2002.

[25] Kaaniche M., Laprie J.C., and Blanquart J.P., "Dependability Engineering of Complex Computing Systems", IEEE, 2000.

[26] Kazman R., http://www.sei.cmu.edu/news-at-sei/columns/the_architect/1999/June/Architect.jun99.pdf, Using scenarios in architecture evaluations.

[27] Kazman R., Kruchten P., Nord R.L., and Tomayko J.E., Integrating Software-Architecture-Centric Methods into the Rational Unified Process, report CMU/SEI-2004-TR-011, Software Engineering Institute SEI, 2004.

[28] Kazman R., Nord R.L., and Klein M., A Life-Cycle View of Architecture Analysis and Design Methods, report CMU/SEI-2003-TN-026, Software Engineering Institute SEI, 2003.

[29] Kotonya G. and Sommerville I., *Requirements Engineering - Processes and Techniques*, John WIley & Sons Ltd, 1997.

[30] Laprie J.-C., e., Dependability Handbook, report 98-346, Laboratory for Dependability Engineering LAAS, 1998.

[31] Liu J.W.S and others, Imprecise computations, *Proceedings of the IEEE*, volume 82, issue 1, 1994.

[32] Maier M. and Rechtin E., *The art of systems architecting*, CRC Press, 2000.

[33] Microsoft Corporation, http://msdn.microsoft.com/library (Feb 2003), How Windows CE .NET is Designed for Quality of Service.

[34] Microsoft Corporation, *Windows Hardware and Driver Central homepage*, http://www.microsoft.com/whdc/, 2003.

[35] Microsoft Research, http://research.microsoft.com/slam/, SLAM Project 2003.

[36] Musa J.D., Operational profiles in software-reliability engineering, *IEEE Software*, volume 10, issue 2, 1993.

[37] Mustapic G. and Crnkovic I., "Propagation of quality attributes in a layered design", Third Conference on Software Engineering Research and Practise in Sweden, SERPS'03, Lund, Sweden, 2003.

[38] Mustapic G. and et al, *A Dependable Open Platform for Industrial Robotics - A Case Study*, in Rogerio de Lemos A.R.a.C.G. (editors): *Architecting Dependable Systems II*, ISBN 3-540-23168-4, Springer, 2004.

[39] Mustapic G. and et al, "Real World Influences on Software Architecture - Interviews with Industrial Systems Experts", IEEE Working Conferance on Software Architectures, WICSA, Oslo , June 2004, 2004.

[40] Mustapic G., Wall A., Norstrom C., Crnkovic I., Sandström K., Fröberg J., and Andersson J., http://www.idt.mdh.se, MRTC Technical Report: Influences between Software Architecture and its Environment in Industrial Systems - a Case Study.

[41] Neill C.J. and Laplante P.A., Requirements Engineering: The State of the Practice, *IEEE Software*, volume 20, issue 6, 2003.

[42] Nord R.L., Barbacci M., Clements P., Kazman R., Klein M., and et al, Integrating Architecture Tradeoff Analysis Method (ATAM) with the Cost Benefit Analysis Method (CBAM), report CMU/SEI-2003-TN-038, SEI Software Engineering Institute, Carnegie Mellon University, PA USA, 2003.

[43] Nord R.L., Wood G.W., and Clements P., Integrating Quality Attribute Workshop (QAW) and the Attribute-Driven Design (ADD) Method, report CMU/SEI-2004-TN-017, SEI Software Engineering Institute, Carnegie Mellon University, PA USA, 2004.

[44] Parnas D., "The Limits of Empirical Studies of Software Engineering", IEEE International Symposium on Empirical Software Engineering, ISESE, 2003.

[45] Pfleeger S.L., *Software Engineering - Theory and Practice*, Prentice Hall, 2000.

[46] Pfleeger S.L. and Kitchenham B., Software Quality: The Elusive Target, *IEEE Software*, 1996.

[47] Poort R.E. and With H.N.P, "Resolving Requirements Conflicts through Non-Functional Decomposition", WICSA'04 4th Working IEEE/IFP Conference on Software Architecture, Oslo, Norway, 2004.

[48] Preiss O., "Foundations of Systems and Properties: Methodological Support for Modeling Properties of Software -Intensive Systems", Ecole Polytechnique Federale de Lausanne, Switzerland, 2004

[49] Preiss O. and Wegmann A., "Stakeholder discovery and classification based on systems science principles", Second Pacific-Asia Conference on Quality Software, 2001.

[50] Raadt B. and et al, "Polyphony in Architecture", Proceedings of the 26th International Conference on Software Engineering (ICSE'04), 2004.

[51] Robson C., *Real World Research, second edition*, Blackwell Publishers, 2002.

[52] Schwanke W.R., "Architectural Requirements Engineering: Theory vs. Practice", STRAW'03 Second International SofTware Requirements to Architectures Workshop, 2003.

[53] Shaw M., "The Coming-of-Age of Software Architecture Research", 23rd International Conference on Software Engineering, ICSE, 2001.

[54] Shaw M., "Writing good software engineering research papers: minitutorial", Proceedings of the 25th International Conference on Software engineering, ICSE, 2003.

[55] Smolander K., Hoikka K., Isokallio J., Kataikko M., and Mäkelä T., "What is Included in Software Architecture? – A Case Study in Three Software Organizations",

proceedings of the 9th IEEE International Conference and Workshop on Engineering of Computer-Based Systems, 2002.

[56] Sommerville I., *Software Engineering*, Addison-Wesley, 2001.

[57] Soni D., Nord R.L., and Hsu L., "An Empirical Approach to Software Architectures", proceedings of the 7th International Workshop on Software Specification and Design, 1993.

[58] Takashio K. and Tokoro M., "An Object-Oriented Language for Distributed Real-Time Systems", OOPSLA'92, Vancouver, 1992.

[59] Thane H., "Monitoring, Testing and Debugging of Distributed Real-Time Systems", Doctoral Thesis, Royal Institute of Technology, KTH, Mechatronics Laboratory, TRITA-MMK 2000:16, Sweden, 2000

[60] Thane H., Sundmark D., Huselius J., and Pettersson A., "Replay Debugging of Real-Time Systems using Time Machines", Parallel and Distributed Processing Symposium, 2003.

[61] Voas J., Trusted Software's Holy Grail, *Software Quality Journal*, volume 11, issue 1, 2003.

[62] Wall A., Andersson J., and Norstrom C., "Probabilistic Simulation-based Analysis of Complex Real-Time Systems", 6th IEEE International Symposium on Object-oriented Real-time distributed Computing, Hakodate, Hokkaido, Japan, 2002.

[63] Yao P., http://msdn.microsoft.com/library (Sep 2002), Choosing a Windows Embedded API: Win32 vs. the .NET Compact Framework.