

Blekinge Institute of Technology
Dissertation Series No 2002 - 2



Architecture-Level Modifiability Analysis

PerOlof Bengtsson

Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Sweden 2002

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2002-2
ISSN 1650-2159
ISBN 91-7295-007-2

Architecture-Level Modifiability Analysis

PerOlof Bengtsson



Department of Software Engineering and Computer Science
Blekinge Institute of Technology
Sweden

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2002-2

ISSN 1650-2159
ISBN 91-7295-007-2
Published by Blekinge Institute of Technology
© 2002 PerOlof Bengtsson

Cover photography, "The Golden Gate Bridge"
by PerOlof Bengtsson © 1997, all rights reserved.

Printed in Sweden
Kaserntryckeriet, Karlskrona 2002

*"Real knowledge is to know the
extent of one's ignorance."
Confucius*

This thesis is submitted to the Faculty of Technology at Blekinge Institute of Technology, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Software Engineering.

Contact Information:

PerOlof Bengtsson
Department of Software Engineering
and Computer Science
Blekinge Institute of Technology
Box 520
SE-372 25 RONNEBY
SWEDEN
email: perolof.bengtsson@swipnet.se

Abstract

Cost, quality and lead-time are three main concerns in software engineering projects. The quality of developed software has traditionally been evaluated on completed systems. Evaluating the product quality at completion introduces a great risk of wasting effort on software products with inadequate system qualities. It is the objective of this thesis to define and study methods for assessment, evaluation and prediction of software systems' modifiability characteristics based on their architecture designs. Since software architecture design is made early in the development, architecture evaluation helps detect inadequate designs and thus reduces the risk of implementing systems of insufficient quality.

We present a method for architecture-level analysis of modifiability (ALMA) that analyses the modifiability potential of a software system based on its software architecture design. The method is scenario-based and either compares architecture candidates, assesses the risk associated with modifications of the architecture, or predicts the effort needed to implement anticipated modifications. The modification prediction results in three values; a prediction of the modification effort and the predicted best- and worst-case effort for the same system and change scenario profile. In this way the prediction method provides a frame-of-reference that supports the architect in the decision whether the modifiability is acceptable or not.

The method is based on the experiences and results from one controlled experiment and seven case-studies, where five case studies are part of this thesis. The experiment investigates different ways to organize the scenario elicitation and finds that a group of individually prepared persons produce better profiles than individuals or unprepared groups.

Acknowledgements

The work presented in this thesis was financially supported by Nutek and the KK-foundation's program, 'IT-lyftet'.

First, I want to express my gratitude to my advisor and friend, *Prof. Jan Bosch*, for giving me this opportunity in the first place and gently pushing me through this process. Thanks to all my colleagues in our research group, especially *Dr. Michael Mattsson*, for tremendous support and inspiring ideas. I would also like to thank *Prof. Claes Wohlin* for valuable comments and help. For fruitful cooperation I thank *Dr. Daniel Häggander*, *Prof. Lars Lundberg*, *Dr. Nico Lassing* and *Prof. Hans van Vliet*.

I would also like to thank the companies that made it possible to conduct the case studies, Althin Medical, Cap Gemini Sweden, DFDS Fraktarna AB, EC-Gruppen AB, and Ericsson for their cooperation. I would especially like to thank *Lars-Olof Sandberg* of Althin Medical; *Joakim Svensson* and *Patrik Eriksson* of Cap Gemini Sweden; *Stefan Gunnarsson* of DFDS Fraktarna; *Anders Kambrin* and *Mogens Lundholm* of EC-Gruppen AB; *Åse Petersén*, *David Olsson*, *Henrik Ekberg*, *Stefan Gustavsson*, and *Staffan Johnsson* of Ericsson for their valuable time and input. Philips Research and the SwA-group for having me there and their kind understanding at difficult times.

Many students of the software engineering curriculum at Blekinge Institute of Technology have contributed their valuable time and effort to this research and for that you have my sincere gratitude.

Thanks to all of my friends and neighbors for the many enjoyable alternatives to writing this thesis.

My whole family has been essential during this time, in our moments of grief as well as our moments of joy and I thank you all; my father *Hans*, my brother *Jonas* and his partner *Åse* and their lovely children, my niece and nephew, *Petronella* and *Gabriel*, and my sister *Elisabeth*. Thanks also to my extended family, the *Berglund/Alexanderssons*. Finally, thank you, *Kristina*, your love and understanding made this enterprise endurable.

PerOlof Bengtsson, 9th of January, 2002, Ronneby, Sweden.

Table of Contents

INTRODUCTION 1

Research Questions 3

Research Methods 5

Research Results 13

Further Research 14

Related Publications 15

PART 1: THEORY

CHAPTER 1: SOFTWARE ARCHITECTURE 19

Software Architecture Design 23

Software Architecture Description 34

Software Architecture Analysis 44

Conclusions 51

CHAPTER 2: MODIFIABILITY ANALYSIS 53

Modifiability 54

Analyzing Modifiability 55

Architecture-Level Modifiability Analysis Method (ALMA) 56

Conclusions 62

CHAPTER 3: MODIFIABILITY PREDICTION MODEL 65

Scenario-based Modifiability Prediction 65

Best and Worst Case Modifiability 67

Conclusions 71

CHAPTER 4: SCENARIO ELICITATION 73

Change Scenarios 73

Change Scenario Profiles 74

Elicitation Approaches 76

Conclusions 80

CHAPTER 5: AN EXPERIMENT ON SCENARIO ELICITATION 83

Experiment Design 84

Threats 91

Analysis & Interpretation 94

Analysis Based on Virtual Groups 101

Conclusions 106

CHAPTER 6:	HAEMO-DIALYSIS CASE	111
	Lessons Learned	116
	Architecture Description	121
	Evaluation	131
	Conclusions	133
CHAPTER 7:	BEER-CAN INSPECTION CASE.....	135
	Goal Setting	137
	Architecture Description	137
	Change Scenario Elicitation	138
	Change Scenario Evaluation	140
	Interpretation	148
	Conclusions	150
CHAPTER 8:	MOBILE POSITIONING CASE.....	151
	Goal Setting	152
	Architecture Description	152
	Change Scenario Elicitation	154
	Change Scenario Evaluation	157
	Interpretation	158
	Conclusions	160
CHAPTER 9:	FRAUD CONTROL CENTRE CASE.....	161
	Goal Setting	162
	Architecture Description	163
	Change Scenario Elicitation	164
	Change Scenario Evaluation	164
	Interpretation	165
	Conclusions	166
CHAPTER 10:	FRAKTARNA CASE.....	167
	Goal Setting	168
	Architecture Description	168
	Change Scenario Elicitation	170
	Change Scenario Evaluation	172
	Interpretation	175
	Conclusions	176
CHAPTER 11:	SOFTWARE ARCHITECTURE ANALYSIS EXPERIENCES	177
	Experience concerning the analysis goal	177
	Architecture description Experiences	178
	Scenario elicitation Experiences	179
	Scenario evaluation Experiences	182
	Experiences with the results interpretation	184
	General experiences	185
	Conclusions	186
	CITED REFERENCES	188
	AUTHOR BIOGRAPHY	196
APPENDIX A:	INDIVIDUAL INFORMATION FORM.....	197
APPENDIX B:	INDIVIDUAL SCENARIO PROFILE	198
APPENDIX C:	GROUP SCENARIO PROFILE	199
APPENDIX D:	VIRTUAL GROUP RESULTS	200

Introduction

Cost, quality and lead-time are three main concerns that make software engineering projects true challenges. Cost should be low to increase profit, quality should be high to attract and satisfy customers and lead-time should be short to reach the market before the competitors. The pressure to improve cost, quality and lead-time in order to stay in business is perpetual.

Cost, quality and lead-time are not independent from each other, improvements in one factor, may affect at least one of the others negatively. For example, adding more staff to a project can sometimes decrease the lead-time but increases the costs, or, spending more time on testing may increase the quality but also increases costs. The problem of managing software development is therefore an optimization problem. The goal is to reach an optimum of cost versus quality versus lead-time and it requires control over the development process as well as the design.

Software product quality was often used to mean the absence of faults in the delivered program code. The meaning and understanding of software product quality has shifted and software product quality has been defined in the ISO/IEC FDIS 9126-1 standard to mean the many different characteristics of the software system. For example, dependability, data throughput, modifiability, or response time. The problem of controlling the software quality in the development process, is to know if the chosen solution will achieve the required software quality, concerning, for example, data throughput or software maintainability.

The quality of developed software have traditionally been evaluated on the completed system just before delivering the system to the customer. The risk that large efforts have been spent on developing systems that do not meet the quality requirements is obvious. In the event of such failure, there are

really only two principal options; to abandon the system and cut the losses, or, to rework the system until it does meet the requirements at even more expenses.

To modify the software design of a software system once it has been implemented in source code very likely requires major reconstruction of the system. Because we still lack ways to evaluate the design against the quality requirements, we again face the risk of completing a system that does not meet the requirements. As if the budget over-run was not problem enough. While, of course, we gain valuable experiences from the first failure we still run a real risk of failing again. Although this is a less attractive way to develop software, it can be observed as current practise in many software developing organizations.

The consequences of building a software system that does not meet the required qualities range from financial losses to fatalities. Anyone who has tried to access, for example, an internet shop using first a high-speed network and then a standard (V.90) phone modem, have experienced the value of response time. Although it is the same web site with the logically same functions in both cases, it is less valuable to use in the latter case, due to the longer response times. Such differences in software system qualities may mean serious financial problems because of lost business. In many cases, such as the Internet-example, a strict limit for what is acceptable is very hard to find. Some persons are more patient than others. In other cases, failing to meet the quality requirements may be fatal. Haemo dialysis machines is only one example where human life may be jeopardized. Should, for example, the software system in such a machine fail to respond fast enough to sudden blood pressure changes, the patients life may be at serious risk.

Further, we desire several different qualities from systems, e.g. high reliability, low latency and high modifiability. The qualities we desire from the system are all aspects of the *same* system and hence not completely independent from each other. The result is that a solution intended to improve a certain quality of the system, will most likely also affect other qualities of the system. For example, adding code that makes the system flexible, also means that there are more instructions to execute and thus adds to the response times in certain situations. Hence, the challenge facing the software architect is to find a

design that balances the software qualities of the system, such that all quality requirements on the system are met.

Achieving such balance of qualities of the software system requires not only the techniques to estimate each quality attribute of the to be completed system, but also systematic methods for how to arrive at a well balanced design. Also, the earlier in the development process we can make use of these techniques and methods, the more we may eliminate the risk of wasting resources on flawed designs.

Almost thirty years ago Parnas (1972) argued that the way a software system is decomposed into modules affects its abilities to meet levels of efficiency in certain aspects, e.g. flexibility and performance. Software architecture is concerned with precisely this, *viz.* the structure and decomposition of the software into modules and their interactions. It is commonly held in literature (Bass *et al.* 1998; Bosch 2000; Bushmann *et al.* 1996; Garlan & Shaw 1996), that the software architecture of a system sets the boundaries for its qualities. Most often, the software architecture embodies the earliest design decisions, before much resources have been put into detailed design and source code implementation. Indeed does this make software architecture design and evaluation promising areas for addressing the challenges described above. It is fair to say that the development organization that has the capability to assess their earliest designs', i.e. the software architecture, potential to meet the quality requirements, also has an important competitive advantage over those competitors who lacks the ability. This thesis is about obtaining a satisfactory level of such control through the use of a systematic software architecture design method and scenario-based modifiability analysis of software architecture.

Research Questions

This thesis focus on scenario-based modifiability analysis of software architecture and the prediction of the completed system's modifiability. It includes the context in which such methods are used: the information needed in terms of software architecture descriptions and documentation; the processes and techniques to perform such analysis; the possible interpretation of the analysis results.

We have deliberately limited the work to address modifiability analysis. Analysis of different quality attributes have their own specific problems and possibilities. A complete software architecture assessment demands that other quality attributes are analyzed as well.

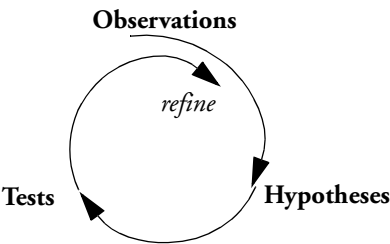
We identified four types of software architecture analysis, or situations in which you need software architecture analysis to make a decision. These are the typical situations:

- 1 Architecture A versus architecture B
This is the situation when the candidates are of different origins and their designs differ significantly. The decision to make is which one to use, and requires that we find out if candidate A is better than B, or not.
- 2 Architecture A versus architecture A'
This is the situation when the candidates are subsequent versions or two versions stemming directly from the same ancestor. The decision is again a relative one, but the close relation and small difference between the candidates provide opportunities for simplifying the comparison, e.g. by reducing the scope of the analysis based on the changes made.
- 3 Architecture A versus a quantified quality requirement.
This is the situation when we want to know not only if we have chosen the best available candidate, but also if it will meet the requirements. This calls for a method that can translate the software architecture design into the same scale as the requirement has been expressed.
- 4 Architecture A versus architecture X
This is the situation when we want to know if there, in theory, could exist a software architecture design that is even better with respect to the particular quality. In a way, it is to benchmark against a virtual competitor. The decision to continue improving the design is clearly dependent on the room to improve certain aspects.

The aim of this research has been to propose methods that support these four typical situations.

The research process can be seen as an refinement cycle (figure 1). First, we study the world around us. From these observations we form theories or hypotheses on the nature of the world. We test those hypotheses by carefully manipulating the world and observing the results, which puts us right back where we started in the cycle. Each completed cycle teaches us something about the world.

Figure 1.
*The research
refinement cycle*



The findings and conclusions in this thesis are the results of applying a number of different research methods. In part II the results are based on case study/action research. In chapter 5 we performed a controlled experiment. The remainder of this section discusses those methods.

Case studies

Robson (1993) advocates the use of case studies as a valid and acceptable empirical scientific method. He provides the following definition of what a case study is:

Case study is a strategy for doing research which involves an empirical investigation of a particular phenomenon within its real life context using multiple sources of evidence.
(Robson, 1993, page 5)

**Designing Case
Studies**

Robson (1993) describes four corner stones of case study design; the conceptual framework, the research questions, the sampling strategy, and the methods and instrument for collecting data.

The *conceptual framework* involves describing the main features of the phenomenon of the studies, e.g. aspects, dimensions,

factors and variables. An important part is also the known or presumed relationships between these.

The *research questions* are important to be able to make a good decision about the data to be collected. A common way is to use the conceptual framework as a basis for formulating the research questions. However, some might prefer the other way around.

The *sampling strategy* basically means answering the questions, who? where? when? what? and make a decision of how to make the sample. Some initial options in sampling include; settings, actors, events, and processes, which could be used as a starting point. The author gives the hint that it is generally the case that whatever sampling plan that is decided upon, will not be possible to complete in full.

The *methods and instruments for collecting data* is dependent of the type of results expected. Either it could be exploratory in nature, i.e. basis for inductive reasoning and hypothesis building, or, it could be confirmatory in nature, i.e. finding data that supports the given hypothesis.

Multiple Case Studies

Conclusions from case studies are not as strong as a controlled experiment, and it is claimed that the use of multiple cases yields more robustness to the conclusions from the study (Robson, 1993; Yin 1994). The reason for this is not, as the quantitatively oriented researcher might assume, that the sample is bigger. Instead, the reasons lie in other important aspects.

First, multiple case studies distinguish themselves from, for example, surveying *many* persons about something instead of one, or, increasing the number of subjects *within* an experiment. Instead the usage of multiple cases should be regarded similar to the replication of an experiment or study. This means that the conclusions from one case should be compared and contrasted with the results from the other case(s).

Second, the number of cases needed to increase the sample and the statistical strength, would require more cases than what is probably afforded or even available. Instead the selection of the cases for multiple case study is categorized into two types of selection. The *literal replication* means that the cases selected are similar and the predicted results are similar too. The *theoretical*

replication means that the cases are selected based on the assumption that they, according to the tested theory, will produce opposite results.

Multiple case studies is to prefer over single case studies in most situations to achieve more robust results. There are, however, a few situations where the multiple case study is not really applicable per definition, or because it offers little or no improved robustness to the results. These situations are; the extreme and unique case, the critical case, and the revelatory case.

- The *extreme and unique case* is the phenomenon that is so rare, or extreme, that any single case is worth documenting.

Sometimes researchers have the opportunity to study extreme or unique cases of the phenomenon they are interested in. For example, the recovery of a severely injured person that under normal circumstances would not survive. In this case, the issue is not generalization but should be compared more to the *counter example* proof in logic. It is used to falsify hypotheses, by showing one counter example. In medicine, for example, an unique case could, in the battle against viral deceases, be a survivor from a very deadly virus infection.

- The *critical case* is the one case that may challenge, confirm or extend the hypothesis formulated.

In many theories a researcher can identify a particular case which would either make or break the theory. In these situations it is not, per definition, very necessary to use a multiple case design. Instead if we actually have identified the critical case, we need only to investigate that particular case.

- *Revelatory case* first opportunity to study phenomenon.

The revelatory case is often much appreciated among researchers in the field of the study. Per definition it is the first time the phenomenon is studied, or revealed. The goal of such a study is not primarily to validate hypotheses, since in these cases existing hypotheses are generally very weak. Instead the goal is to explore the phenomenon that was never studied before.

Strengths Case studies are less sensitive to changes in the design during the implementation of the study than experiments are, because less control is required. Unless the researcher are very lucky the assumptions made in the case study plan will not all hold. Mostly, when such deviations from the plan do occur in a case study, one can still meaningfully interpret the results, as opposed to experiments that can be severely damaged.

Case studies are also more suitable than experiments for collecting and analyzing qualitative data. Software engineering research often involves social and human factors that are hard to meaningfully quantify on measurement scales.

Limitations Case studies major limitation is that the results are not generalizable to the same extent as for randomized experiments.

Action Research

Action research is an established research method in medical and social science since the middle of the twentieth century. During the 1990's the action research method has gained interest in the information systems research community and several publications based on the results research testify to this.

In action research one can distinguish four common characteristics (Baskerville, 1999):

- an action and change orientation
- a problem focus
- a systematic and sometimes iterative process in stages
- collaboration among participants.

In its most typical form, action research is a participatory method based on a five step cyclical model (Baskerville, 1999):

- 1 Diagnosing is to come to an understanding of the primary problem and come to a working hypothesis.
- 2 Action planning is the collaborative activity of deciding what activities should be done and when they should be done.
- 3 Action taking is to implement the plan. This can be done in different levels of intervention from the researcher.
- 4 Evaluating is the collaborative activity of determining whether the theoretical effects.

- 5 Specifying learning is the activities concerned with reporting and disseminating the results.

This five step process is similar to the general view on research presented in figure 1.

Strengths

The basis for action research is that the researcher both participates in and observes the process under study. There are two assumptions that motivates this (Baskerville, 1999). First the (social) situation studies cannot be reduced to a meaningful study object. Second, the participation of the researcher increases the understanding of the problem, i.e. action brings understanding. The consequence is that the researcher generally gets a more complete and balanced understanding of the research issues. Assuming that irrelevant research is in many cases caused by incomplete understanding or misunderstanding of the research problem, then action research should render more relevant research results based on the increased problem understanding.

Additional motivations for using action research is less scientific in nature, but more a question of ethics. Action research allows for a much higher degree of technology transfer during the cooperation with industry than other kinds of research methodologies. This is especially important in fast paced knowledge intensive businesses, e.g. software development companies, or pharmaceutical companies.

Limitations

Action research is clearly not valid in the positivistic sense. This is a limitation of the method such that it make it harder to gain acceptance for the results of the studies. The qualitative nature and interpretative foundations make reporting on such studies spacious and difficult to fit into the general templates of journal and conference articles.

Although there are clear differences action research sometimes may be taken for consulting. This is a ethical problem especially if the partner in the research project expects you to deliver proprietary results, as opposed to writing public research reports.

Controlled Experiments

Experimentation is a powerful means to test hypotheses. Scientific experimentation builds on a set of founding principles. The idea is that while maintaining full control over

certain aspects of the world and carefully observing other aspects, we can test whether our understanding about cause and effect corresponds to the real world study subject. The basic principle is that of logics. Consider a statement that applies for all instances of a population, 'for all X holds P' (hypothesis 1).

$$\forall x | P |$$

Now, if this is true, the opposite must be false. And this is what we will make use of in experimentation. To get the opposite we consider an equivalent to the sentence of our hypothesis. It is the sentence 'There exists no one X such that P is false'.

$$\neg \exists x | \neg P |$$

Now we negate this sentence to find the logical reverse of our hypothesis and get the null hypothesis 'There exists an X such that P is false'.

$$\exists x | \neg P |$$

This way, one, and only one, of the hypothesis and the null-hypothesis must be true. The idea is to design the experiment such that we can reject the null hypothesis and thus conclude that the hypothesis is true. Should not the null hypothesis be the logical reverse of the hypothesis we get invalid results.

Experiment Control

The power of experiments are dependent on the level of control we can achieve. The effort to perform experiments increases with the number of factors that are involved with the phenomenon that we study. In software engineering, there are often several factors involved. In addition these factors are often hard to quantify such that they become relevant and controllable. For example, how can you measure a persons competence such that you can make sure that variations in competence was the reason for the observed effects rather than the treatment?

The general design principles to achieve control and eliminate threats are; randomization, balancing, and blocking (Wohlin *et al.*, 2000).

Statistics is often based on the assumption that every outcome has the same probability of occurrence, i.e. being random. This is utilized especially for selections in the experiment. The principle is that a factor that may affect the outcome has the same probability of doing it positively or negatively. Thus, over a larger set of outcomes, the factors will average out the effect.

Balancing is to make sure that each treatment is assigned the same number of subjects. It simplifies and strengthens the statistical analysis.

Sometimes we are not interested in a factor that we know to affect the outcome. In such cases we can use blocking. Blocking means that we group, or *block*, the subjects such that each subject within in a group is affected by the factor in the same way, or equally much. This requires the ability to observe and quantify the factor. The analysis is then kept between the subjects in the same block and no analysis is made between the blocks.

Process The Experiment process consists of the following conceptual steps (Basili *et al.*, 1986):

- 1 Definition
- 2 Planning
- 3 Operation
- 4 interpretation

In their book on experimentation in software engineering, Wohlin *et al.* (2000), adds analysis to the fourth step and the following fifth step.

- 5 Presentation and package

The planning activity in the experiment process requires careful planning of all the following steps, since it is often the case that once the operation step is started, changes to the experiment may make it invalid.

Type I & II Error When testing our hypotheses, we are concerned with two types of error. The first type, Type-I-error, is that we reject the null hypothesis when we should not have. In other words, our analysis show a relationship that does not exists in reality. The second type of error, Type-II-error, is that we do not reject the null hypothesis when we should have.

Because the validity of the conclusions from experiments are dependent on rigid control, it is crucial to consider anything that might jeopardize the experiment to be a threat. The experiment validity may be divided into four types of validity, each which have its own type of threats. The four types are; internal validity; conclusion validity; external validity; construct validity (Wohlin *et al.*, 2000).

Internal validity In the experiment design and operation we must make sure that the treatment and only the treatment causes the effects we observe as the outcome.

Conclusion validity In the experiment analysis and interpretation we must make sure that the relationship between the treatment and the outcome exists. Often this means establishing a statistical relationship with a certain degree of significance.

External validity When interpreting the results from the experiment we must be careful when generalizing the results beyond the study itself. Randomized sampling of the experiment subjects is often used to allow the conclusion from the experiment to extend over a larger population than participating the experiment itself.

Construct validity When designing the experiment we must make sure that the observations we plan detect only that which is an effect of the treatment according to the theory and hypothesis.

Experiments can be divided into categories based on their scope. Basili *et al.* (1986) defines the categories in table 1. Generally, the blocked-subject project design is the best with respect to control and external validity. Such designs eliminate some important threats, e.g. learning effects between projects.

Table 1: Scope of experiment designs

#Teams per project	#Projects	
	one	more than one
one	Single project	Multi-project variation
more than one	Replicated project	Blocked subject-project

Experiment replication Because the results from experiments are very sensitive to a plethora of validity threats, we need replication. Replication is the repeated independent execution of an experiment design, with a different sample of subjects. The goal is to get independent confirmation of the conclusions.

Power The power of a statistical test is the probability that the test will reveal a true pattern when it exists in reality. Because the execution of experiments often cost a lot of time and money, it can help when designing a cost-effective experiment to calculate the statistical power of the design beforehand .

In relation to the research questions we have made the following contributions:

Concerning the comparison of architecture candidates, be it architecture A versus architecture B, or architecture A versus A' we have presented a method and a case study showing how this can be done in practise (see chapter 7). In its current form the method does not make use of the difference in the two types of comparison, i.e. two completely different candidates and comparing the architecture to the proposed successor in the iterative design process.

Concerning the assessment whether an architecture A will provide the potential for the system to meet a modifiability quality requirement we have presented a method for prediction the modifiability effort given a change scenario profile (Chapter 3). Indeed, there are other quality requirements and modifiability requirements can perhaps be defined in other terms than required effort, but that remains for future work.

The last of the research questions concerns the issue of reference when interpreting the results of an analysis. It is really an issue common to all kinds of predictions and evaluations. Concerning the comparison of an architecture candidate A to a hypothetical best- and worst-case architecture, we have, based on two explicit assumptions, presented a model and technique for providing the comparison (Chapter 3).

In addition to the research questions that have guided this work, we have also contributed the following results:

- the elicitation process and its impact on the analysis results (Chapter 4). We have shown that the elicitation process is crucial to the interpretation of the results of the analysis.
- the impact of individuals and groups on the change scenario profile using an experiment (Chapter 5). We found that there are strong support for using a group of individuals that prepared their own change scenario profiles and then merge them into one in a meeting.
- experiences from the analysis process that show the bias and variation in views different stakeholders have (Chapter 11).

The results in this thesis raises new questions within the architecture-level modifiability analysis and architecture assessment research fields. There are primarily four issues that requires further studies; the accuracy of the predictions, the assumptions made in the best and worst case model, how the created scenario profile can be verified, and if & how the prediction model needs to be adapted to each case of use.

Accuracy In the case of modifiability predictions an intuitive way of studying the accuracy would be to; study several projects and architectures; perform early predictions using the method and then collecting the measures of modification and time during the later stages of evolution. The problem is that it takes years to perform and to find a large enough set of study subjects to reach generalizable results is a major challenge. The issue of accuracy is relevant to many more of the architecture analysis methods presented in literature. In order to really move the area forward there is a need for other approaches to determining the accuracy. For example, to perform controlled laboratory experiments to verify the underlying assumptions in these methods.

Assumptions In comparison to the modifiability prediction model, the best- and worst-case modifiability prediction model is based on two additional assumptions; differences in productivity and modification size invariance (see “Best and Worst Case Modifiability” on page 67). In our work we have searched in literature to find research results that could corroborate or contradict these assumptions. The limitation seem to be that research concerning productivity and modification size has been performed within a project or otherwise different angle, e.g. to predict maintenance effort predictions based on code metrics (Li and Henry, 1993). Therefore, it is a necessary step to empirically study whether these assumptions hold.

Scenario Verification As identified in this thesis the scenario profiles play an important role for the accuracy of the results in all scenario based methods. Although we have addressed the issue of verifying the change scenario profile used in the prediction method, we believe that more effort needs to be spent on this topic. The robustness of scenario-based method could be greatly improved if they were to be accompanied by techniques for verifying that the scenario profile is valid in each respective

case. The scenario profile needs to be complete and correct and both these issues must be addressed by a verification technique. The opportunities for finding such verification techniques differ with the nature of the quality factor which the scenario profile is supposed to address. In modifiability the profile addresses future needs and events, and present verification challenges are much different from those of a usage profile used to investigate system performance.

Model Adaptation

The prediction models presented in this thesis are based on implicit and explicit assumptions. Some of these assumptions are based on the understanding of processes and products in the domains in which the case studies have been performed. It is possible that the accuracy of the prediction can be improved by modifying the prediction model to more closely adhere to the product and processes in different domains. Such investigations are related to the problem of establishing the accuracy of the prediction, but aims at finding what alterations could be made and guidelines for when to make them.

In addition to the issues discussed above, the results and studies need independent replication and verification by other software engineering researchers.

Related Publications

This monographic thesis is based on a number of refereed research articles authored or co-authored by myself. Below is a list of these articles in chronological order:

Paper I Scenario-based Software Architecture Reengineering

PerOlof Bengtsson & Jan Bosch
Proceedings of the 5th International Conference on Software Reuse (ICSR5); IEEE Computer Society Press, Los Alamitos, CA; pp. 308-317, 1998

Paper II Haemo Dialysis Software Architecture Design Experiences

PerOlof Bengtsson & Jan Bosch
Proceedings of ICSE'99, International Conference on Software Engineering; IEEE Computer Society Press, Los Alamitos, CA; pp. 516-525, 1999.

Paper III Architecture Level Prediction of Software Maintenance

PerOlof Bengtsson & Jan Bosch

Proceedings of CSMR'99, 3rd European Conference on Software Maintenance and Reengineering, Amsterdam; pp. 139-147, 1999.

Paper IV Maintainability Myth Causes Performance Problems in Parallel Applications

Daniel Häggander, PerOlof Bengtsson,
Jan Bosch and Lars Lundberg.

In Proceedings of 3rd Annual IASTED International Conference on Software Engineering and Applications (SEA'99), Scottsdale, Arizona, 1999, pp. 288-294.

Paper V An Experiment on Creating Scenario Profiles for Software Change

PerOlof Bengtsson & Jan Bosch

In Annals of Software Engineering (ISSN: 1022-7091),
Bussum, Netherlands: Baltzer Science Publishers, vol. 9,
pp. 59-78, 2000.

Paper VI Analyzing Software Architectures for Modifiability

PerOlof Bengtsson, Nico Lassing,
Jan Bosch, and Hans van Vliet.

Research Report 2000:11, ISSN: 1103-1581, submitted for publication.

Paper VII Experiences with ALMA: Architecture-Level Modifiability Analysis

Nico Lassing, PerOlof Bengtsson,
Hans van Vliet, and Jan Bosch.

Accepted for publication in Journal of Software and Systems, 2001.

Paper VIII Assessing Optimal Software Architecture Maintainability

Jan Bosch, PerOlof Bengtsson,

Proceedings of Fifth European Conference on Software Maintenance and Reengineering (CSMR'01), IEEE Computer Society Press, Los Alamitos, CA, 2001 pp. 168-175.

PART 1:

Theory

This part of the thesis begins with an introduction to software architecture and discusses other authors' contributions related to the work presented in this thesis. Next comes discussions of theories and methods for scenario-based modifiability analysis of software architectures. The focus is especially on that of modification effort prediction including best and worst-case and scenario elicitation. Part I ends with a chapter describing an experiment concerning scenario elicitation and the findings from that experiment.

Part II of this thesis presents case studies on different aspects of the work presented in Part I.

CHAPTER 1: Software Architecture

During the late 1960s and 1970s the concept of systems decomposition and modularization, i.e. dividing the software into modules appeared in articles at conferences, e.g. the NATO conference in 1968 (Randell, 1979). In his classic paper from 1972, David L. Parnas reported on the problem of increasing software size in his article, “On the Criteria To Be Used in Decomposing Systems into Modules” (Parnas, 1972). In that article, he identified the need to divide systems into modules by other criteria than the tasks identified in the flow chart of the system. A reason for this is, according to Parnas, that “The flow chart was a useful abstraction for systems with in the order of 5,000-10,000 instructions, but as we move beyond that it does not appear to be sufficient; something additional is needed”. Further Parnas identifies information hiding as a criterion for module decomposition, i.e. every module in the decomposition is characterized by its knowledge of a design decision which it hides from all modules.

Thirteen years later, in 1985, Parnas together with Paul Clements and David Weiss brings the subject to light again, in the article “The Modular Structure of Complex Systems” (Parnas *et al.*, 1985). In the article it is shown how development of an inherently complex system can be supplemented by a hierarchical *module guide*. The module guide tells the software engineers what the interfacing modules are, and help the software engineer to decide which modules to study.

Shaw and Garlan states that the size and complexity of systems increases and the design problem has gone beyond algorithms and data structures of computation (Shaw and Garlan, 1996). In addition, there are the issues of organizing the system in large, the control structures, communication protocols, physical distribution, and selection among design alternatives. These issues are part of software architecture design.

In the beginning of the 1990:s software architecture got wider attention in the software engineering community and later also in industry. Today, software architecture has become an accepted concept, perhaps most evident in the new role; *software architect*, appearing in the software developing organizations. Other evidence includes the growing number of software architecture courses on the software engineering curricula and attempts to provide certification of software architects.

Elements, form and rationale

In the paper “Foundations for the study of software architecture”, Perry and Wolf (1992) define software architecture as follows:

$$\textit{Software Architecture} = \{\textit{Elements}, \textit{Form}, \textit{Rationale}\}$$

Thus, a software architecture is a triplet of (1) the elements present in the construction of the software system, (2) the form of these elements as rules for how the elements may be related, and (3) the rationale for why elements and the form were chosen. This definition has been the basis for other researchers, but it has also received some critique for the third item in the triplet. It is argued that the rationale is indeed important, but is in no way part of the software architecture (Bass *et al.*, 1998). The basis for their objection is that when we accept that all software systems have an inherent software architecture, even though it has not been explicitly designed to have one, the architecture can be recovered. However, the rationale is the line of reasoning and motivations for the design decisions made by the design, and to recover the rationale we would have to seek information not coded into software. The objection implies that software architecture is an artifact and that it could be coded, although scattered, into source code.

Components & connectors

In a paper about software architecture by Shaw and Garlan (1996) we find the following definition:

Software architecture is the *computational components*, or simply *components*, together with a description of the *interactions* between these components, the *connectors*.

The definition is probably the most widely used, but has also received some critique for the *connectors*. The definition may be

interpreted that components are those entities concerned with computing tasks in the domain or support tasks, e.g. persistence via a data base management system. Connectors are entities that are used to model and implement interactions between components. Connectors take care of interface adaptation and other properties specific to the interaction. This view is supported in, for example, the Unicon architecture description language (Shaw *et al.*, 1995).

Bass et al

A commonly used definition of software architecture is the one given by Bass *et al.* (1998):

The software architecture of a program or computer system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

This definition emphasizes that the software architecture concerns the structure of the system.

IEEE

The definition given by the IEEE emphasizes other aspects of software architecture (IEEE 2000):

Architecture is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment and the principles guiding its design and evolution.

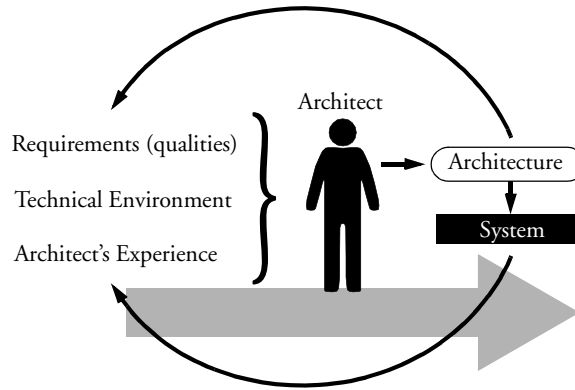
This definition stresses that a system's software architecture is not only the model of the system at a certain point in time, but it also includes principles that guide its design and evolution.

Architecture business cycle

Software architecture is the result from technical, social and business influences. Software architecture distills away details and focuses only on the interaction and behavior between the black box components. It is the first artifact in the life cycle that allow analysis of priorities between competing concerns. The concerns stem from one or more of the stakeholders concerned with the development effort and must be prioritized to find an optimal balance in requirement fulfillment. Stakeholders are persons concerned with the development effort in some way,

e.g. the customer, the end-user, etc. The factors that influenced the software architecture are in turn influenced by the software architecture and form a cycle, the *architecture business cycle* (ABC) (Bass *et al.*, 1998)(figure 2).

Figure 2.
*Architecture
Business Cycle*
(Bass *et al.*, 1998)



In the architecture business cycle the following factors have been identified:

- The software architecture of the built system affects the structure of the organization. The software architecture describes units of the system and their relations. The units serve as a basis for planning and work assignments. Developers are divided into teams based on the architecture.
- Enterprise goals affect and may be affected by the software architecture. Controlling the software architecture that dominates a market means a powerful advantage (Morris and Ferguson, 1993).
- Customer requirements affects and are affected by the software architecture. Opportunities in having robust and reliable software architecture might encourage the customer to relax some requirements for architectural improvements.
- The architect's experience affects and is affected by the software architecture. Architects favor architectures proven successful in the architect's own experience.
- Systems are affected by the architecture. A few systems will affect the software engineering community as a whole.

Software system design consists of the activities needed to specify a solution to one or more problems, such that a balance in fulfillment of the requirements is achieved. A *software architecture design method* implies the definition of two things. First, a process or procedure for going about the included tasks. Second, a description of the results, or type of results, to be reached when employing the method. From the software architecture point-of-view, the first of the aforementioned two, includes the activities of specifying the components and their interfaces, the relationships between components, and making design decisions and document the results to be used in detail design and implementation. The second is concerned with describing the different aspects of the software architecture using different view points.

The traditional object-oriented design methods, e.g. OMT (Rumbaugh et al. 1991), Objectory (Jacobson 1992), and Booch (1999) has been successful in their adoption by companies worldwide. Over the past few years the three aforementioned have jointly produced a *unified modeling language*, UML (Booch et al., 1998) that has been adopted as *de facto* standard for documenting object-oriented designs.

Object-oriented methods describe an iterative design process to follow and their results. When following the prescribed process, there is no way of telling if or when you have reached the desired design results. The reason is that the processes prescribes no technique or activity for evaluation of the halting criterion for the iterative process, i.e. the software engineer is left for himself to decide when the design is finished. This leads to problems because unfinished designs may be considered ready and forwarded in the development process, or, a design that really meets all the desired requirements are not considered ready, for example, because the designer is a perfectionist. In this context is also the problem of knowing wether it is at all possible to reach a design that meet the requirements.

Software architecture is the highest abstraction level (Bass et al., 1998) at which we construct and design software systems. The software architecture sets the boundaries for the quality levels resulting systems can achieve. Consequently, software architecture represents an early opportunity to design for

software quality requirements, e.g. reusability, performance, safety, and reliability.

The design method must in its process have an activity to determine if the design result, in this case the software architecture, has the potential to fulfill the requirements.

The enabling technology for the design phase is neither technological nor physical, but it is the human creative capability. It is the task of the human mind to find the suitable abstractions, define relations, etc. to ensure that the solution fulfills its requirements. Even though parts of these activities can be supported by detailed methods, every design method will depend on the creative skill of the designer, i.e. the skill of the individual human's mind. Differences in methods will present themselves as more or less efficient handling of the input and the output. Or more or less suitable description metaphors for the specification of the input and output. This does not prohibit design methods from distinguishing themselves as better or worse for some aspects. It is important to remember that results from methods are *very dependent* on the skill of the persons involved.

Architecture patterns & styles

Experienced software engineers have a tendency to repeat their successful designs in new projects and avoid using the less successful designs again. In fact, these different styles of designing software systems could be common for several different unrelated software engineers. This has been observed in (Gamma *et al.*, 1995) where a number of systems were studied and common solutions to similar design problems were documented as *design patterns*. The concept has been successful and today most software engineers are aware of design patterns.

The concept has been used for software architecture as well. First by describing *software architecture styles* (Shaw and Garlan, 1996) and then by describing *software architecture patterns* (Bushman *et al.*, 1996) in a form similar to the design patterns. The difference between software architecture styles and software architecture patterns have been extensively debated. Two major view points are; styles and patterns are equivalent, i.e. either could easily be written as the other, and the other view point is, they are significantly different since styles are a categorization of systems and patterns are general

solutions to common problems. Either way styles/patterns make up a common vocabulary. It also gives software engineers support in finding a well-proven solution in certain design situations.

Software architecture patterns impact the system in large, by definition. Applying software architecture patterns late in the development cycle or in software maintenance can be prohibitively costly. Hence, it is worth noting that software architecture patterns provide better leverage considered before program coding has started.

Schlaer & Mellor - Recursive design

The authors of the recursive design method (Schlaer and Mellor, 1997) claims that the following five generally held assumptions should be replaced:

- Analysis treats only the application.
- Analysis must be represented in terms of the conceptual entities in the design.
- Because software architecture provides a view of the entire system, many details must be omitted.
- Patterns are small units with few objects.
- Patterns are advisory in nature

the following five views on software design is suggested instead:

- Analysis *can be performed on any domain*.
- Object oriented analysis (OOA) method does not imply anything about the fundamental design of the system.
- Architecture domain, like any domain, can be modeled in complete detail by OOA.
- OOA models of software architecture provide a comprehensive set of large-scale interlocking patterns.
- Use of patterns is *required*.

Domain analysis

Fundamental for the recursive design method is the domain analysis. A domain is a separate real or hypothetical world inhabited by a distinct set of conceptual entities that behave according to rules and policies characteristic of the domain. Analysis consists of work products that identify the conceptual

entities of a single domain and explain, in detail, the relationships and interactions between these entities. Hence, domain analysis is the precise and detailed documentation of a domain. Consequently, the OOA method must be detailed and complete, i.e. the method must specify the conceptual entities of the methods and the relationships between these entities. The elements must have fully defined semantics and the dynamic aspects of the formalism must be well defined (the virtual machine that describes the operation must be defined).

Application Independent Architecture

The recursive design method regards everything as its own domain. An application independent architecture is a separate domain and deals in complete detail with; organization of data, control strategies, structural units, and time. The architecture does not specify the allocation of application entities to be used in the application independent architecture domain. This is what gives the architecture the property of application independence.

Patterns and Code Generation

The recursive design method includes the automatic generation of the source code of the system and design patterns play a key role in the generation process. Design patterns can be expressed as *archetypes*, which is equivalent to defining macros for each element of the patterns. An archetype is a pattern expressed in the target language with added placeholders for the information held in the instance database. The code generation relies heavily on that the architecture is specified using patterns. Therefore use of patterns is absolutely required.

Design Process

The recursive design process defines a linear series of seven operations; each described in more detail in following sections. The operations are:

- 1 Characterize the system.
- 2 Define conceptual entities.
- 3 Define theory of operation.
- 4 Collect instance data.
- 5 Populate the architecture.
- 6 Build archetypes.
- 7 Generate code.

Activities

Start with eliciting the characteristics that should shape the architecture. Attached to the method is a questionnaire with

heuristic questions that will serve as help in the characterization. The questionnaire brings up fundamental design considerations regarding size, memory usage etc. The information source is the application domain and other domains, but the information is described in the semantics of the system. The results are the system characterization report, often containing numerous tables and drawings.

The conceptual entities and the relationships should be precisely described. The architect selects the conceptual entities based on the system characterization and their own expertise and experience, and document the results in an object information model. Each object is defined by its attributes, which in turn is an abstraction of a characteristic.

The next step in the process is to precisely specify the theory of operation. The authors of the method have found that an informal, *but* comprehensive document works well to define the theory of operation, later described in a set of state models.

In the application domain, a set of entities is considered always present or pre-existing. Collecting instance data for populating the instance database means finding those entities, typically only a few items, e.g. processor names, port numbers etc.

The *populator* populates the architecture by extracting elements from the repository containing the application model and then uses the elements to create additional instances in the architecture instance database. The architecture instance database contains all the information about the system.

The building of archetypes is the part where all the elements in the architecture have to be precisely and completely specified. To completely define an archetype we use text written in the target programming language and placeholders to represent the information from the architecture instance database.

The last operation, that of generating the code requires the implementation of a script, called the system construction engine. This script will generate the code from the analysis models, archetypes and the architecture instance database.

4+1 View Model

The 4+1 View model is intended primarily as a way to organize the description of software architectures (Kruchten, 1996) but also more or less prescribe a design approach. For example, the

fifth view (+1) is a list of scenarios that drives the design method.

Design Process The 4+1 View Model consists of ten semi-iterative activities, i.e. all activities are not repeated in the iteration. These are the activities:

- 1 Select a few scenarios based on risk and criticality.
- 2 Create a straw man architecture.
- 3 Script the scenarios.
- 4 Decompose them into sequences of pairs (object operation pairs, message trace diagram).
- 5 Organize the elements into the four views.
- 6 Implement the architecture.
- 7 Test it.
- 8 Measure it/evaluate it.
- 9 Capture lessons learned and iterate by reassessing the risk and extending/revising the scenarios.
- 10 Try to script the new scenarios in the preliminary architecture, and discover additional architectural elements or changes.

Activities The activities are not specified in more detail by the author (Kruchten, 1995). But some comments are given.

- Synthesize the scenarios by abstracting several user requirements.
- After two or three iterations the architecture should become stable.
- Test the architecture by measurement under load, i.e. the implemented prototype or system is executed.
- The architecture evolves into the final version, and even though it can be used as a prototype before the final version, it is not a throw away.

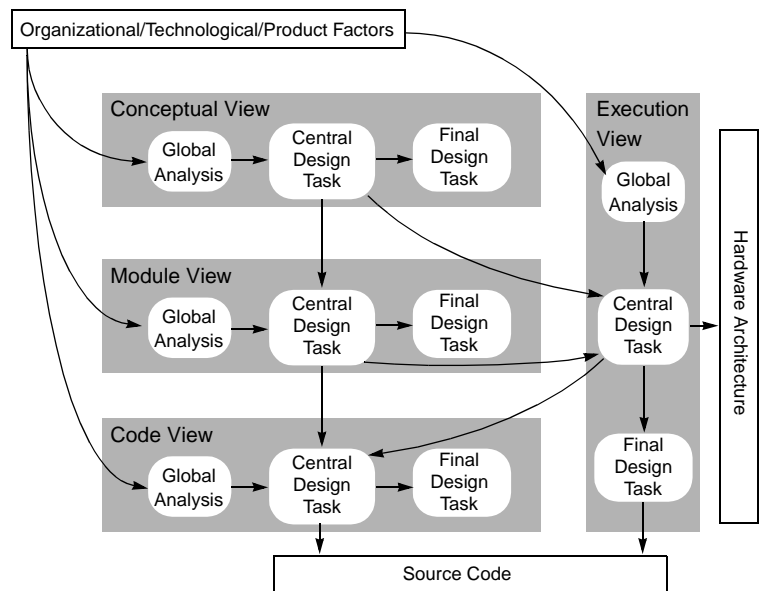
The results from the architectural design are two documents; software architecture views, and a software design guidelines. (Compare to the rationale in the Perry and Wolf definition.)

Hofmeister et al Design Approach

Hofmeister *et al.* (2000) propose an entire approach to designing, describing and analyzing software architectures such that design trade-offs are exposed and handled appropriately. The basis for the design approach is the views; conceptual view, module view, execution view and code view (see description details on page 40).

Design Process The process is connected to the views and the views are to be designed mainly in the order; conceptual view, module view, execution view and code view. Each view has its own three specific main tasks associated as shown in figure 3. The arrows in the figure indicate the main direction of the information flow. Of course, the same flows of feedback occurs.

Figure 3.
Overview of the views and design tasks (Hofmeister et al., 2000)



Activities Each view has specific content in the main tasks and the first activity for each view is the *global analysis* in which you first identify external factors and architecture influencing requirements. Then those factors are analysed with the purpose of deciding on strategies for the architecture design.

The second activity is the *central design task* in which the elements of the particular view and their relationships are defined. The central design tasks typically involve more feedback than the other activities. Within this activity is also the ongoing global evaluation task which does not really produce separate output. It involves deciding on the

information source to use for the evaluation, the verification of the design decisions' impact on prior design decisions.

The final design task is concerned with defining the interfaces and budgeting the resources. This task do not typically influence the other tasks very much.

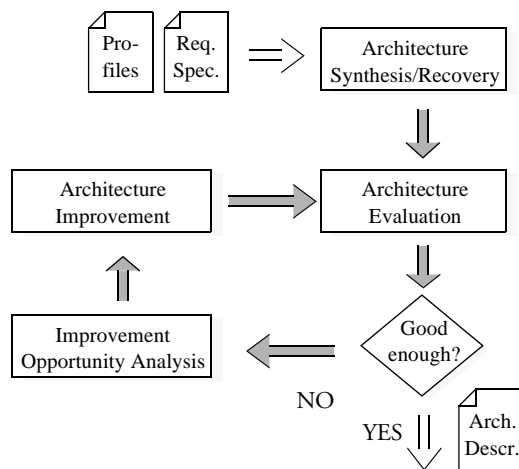
Iterative software architecture design method

The Quality Attribute-oriented Software Architecture (QASAR) design method (Bosch, 2000) exploits the benefits of using scenarios for making software quality requirements more concrete. Abstract quality requirements, like for example reusability, can be described as scenarios in the context of this system and its expected lifetime.

Also, the method puts emphasis on evaluation of the architecture to ensure that the quality requirements can be fulfilled in the final implementation of the system. Four categories of evaluation techniques are described in the method, i.e. scenario-based evaluation, simulation, mathematical modeling and experience-based reasoning.

Design Process The basic process is meant to be iterated in close cycles (figure 4).

Figure 4.
*The basic QASAR
design method
process*



Activities The software architect starts with synthesizing a software architecture design based only on functional requirements. The requirement specification serves as input to this activity. Essentially the functionality-based architecture is the first partitioning of the functions into subsystems. At this stage in

the process, it is also recommended that the scenarios for evaluating the quality requirements be specified. No particular attention is given to the quality requirements, as of yet.

The next step is the evaluation step. Using one of the four types of evaluation techniques the software architect decides if the architecture is good enough to be implemented, or not. Most likely, several points of improvement will reveal themselves during the evaluation and the architecture has to be improved.

Architecture transformation is the operation where the system architect modifies the architecture using the four transformation types to improve the architecture. Each transformation leads to a new version of the architecture with the same functionality, but different quality properties.

Transformations will affect more than one quality attribute, e.g. reusability *and* performance, and perhaps in opposite directions, i.e. improving one and degrading the other. The result is a trade-off between software qualities (McCall, 1994; Boehm and In, 1996). After the transformation, the software engineer repeats the evaluation operation and obtains a new results. Based on these the architect decides if the architecture fulfills the requirements or to iterate.

Evaluation Techniques

The *scenario-based evaluation* of a software quality using scenarios is done in these main steps:

- 1 *Define a representative set of scenarios.* A set of scenarios is developed that concretizes the actual meaning of the attribute. For instance, the maintainability quality attribute may be specified by scenarios that capture typical changes in requirements, underlying hardware, etc.
- 2 *Analyses the architecture.* Each individual scenario defines a context for the architecture. The performance of the architecture in that context for this quality attribute is assessed.
- 3 *Summarize the results.* The results from each analysis of the architecture and scenario are then summarized into an overall results, e.g., the number of accepted scenarios versus the number not accepted.

The usage of scenarios is motivated by the consensus it brings to the understanding of what a particular software quality really means. Scenarios are a good way of synthesising individual interpretations of a software quality into a common view. This

view is both more concrete than the general software quality definition (IEEE 1990), and it is also incorporating the uniqueness of the system to be developed, i.e., it is more context sensitive.

In our experience, scenario-based assessment is particularly useful for development related software qualities. Software qualities such as maintainability can be expressed very naturally through change scenarios. In (Kazman *et al.* 1994) the use of scenarios for evaluating architectures is also identified. The software architecture analysis method (SAAM) however, uses only scenarios and only evaluates the architecture in cooperation with stakeholders prior to detailed design.

Simulation of the architecture (Luckham *et al.* 1995) using an implementation of the application architecture provides a second approach for estimating quality attributes. The main components of the architecture are implemented and other components are simulated resulting in an executable system. The context, in which the system is supposed to execute, could also be simulated at a suitable abstraction level. This implementation can then be used for simulating application behavior under various circumstances. Simulation complements the scenario-based approach in that simulation is particularly useful for evaluating operational software qualities, such as performance or fault-tolerance.

Mathematical modelling is an alternative to simulation since both approaches are primarily suitable for assessing operational software qualities. Various research communities, e.g., high-performance computing (Smith 1990), reliability (Runeson and Wohlin 1995), real-time systems (Liu & Ha 1995), etc., have developed *mathematical models*, or metrics, to evaluate especially operation related software qualities. Different from the other approaches, the mathematical models allow for static evaluation of architectural design models.

A fourth approach to assessing software qualities is through reasoning based on *experience and logical reasoning*. Experienced software engineers often have valuable insights that may prove extremely helpful in avoiding bad design decisions and finding issues that need further evaluation. Although these experiences generally are based on anecdotal evidence, most can often be justified by a logical line of reasoning. This approach is different from the other approaches. First, in that the

evaluation process is less explicit and more based on subjective factors as intuition and experience. Secondly, this technique makes use of the tacit knowledge of the involved persons.

Transformation Techniques

The first category of transformation is to *impose an architectural style*. Shaw and Garlan (1996) and Buschmann *et al.* (1996) present several architectural styles that improve certain quality attributes for the system the style is imposed upon and impair other software qualities. With each architectural style, a fitness for each system property is associated. The most appropriate style for a system depends primarily on its software quality requirements. Transforming architecture by imposing an architectural style results in a major reorganization of the architecture.

A second category of transformations is to *impose architectural patterns*. These are different from architectural styles (Shaw and Garlan, 1996; Buschmann *et al.*, 1996) in that they are not concerned with the main decomposition of the system, but rather cross-cutting functions that is beneficial to factor out, e.g. how to handle concurrency using the periodic object pattern (Molin and Ohlsson, 1998). This kind of transformation is, in a way, similar to aspect oriented programming (Kiczales, 1996). They are also different from design patterns since they affect the larger part of the architecture. Architectural patterns generally impose a *rule* (Richardson & Wolf 1996) on the architecture that specifies how the system will deal with one aspect of its functionality, e.g., concurrency or persistence.

A less dramatic category of transformations is to *apply design patterns* (Gammata *et al.* 1994, Buschmann *et al.* 1996) on a part of the architecture. The application of a design pattern generally affects only a limited number of components in the architecture. In addition, a component can be involved in multiple design patterns without creating inconsistencies.

Another type of transformation is to *convert software quality requirements into functionality*. This solution consequently extends the architecture with functionality not related to the problem domain but is used to fulfil a software quality requirement. Exception handling is a well-known example that adds functionality to a component to increase the fault-tolerance of the component.

Although it is not really a transformation of the architecture, one can, using the divide-and-conquer principle, *distribute a quality requirement* over the subsystems that make up the system. Thus, a software quality requirement X is distributed over the n components that make up the system by assigning a software quality requirement x_i to each component c_i such that $X = x_1 + \dots + x_n$. A second approach to distribute requirements is by dividing the software quality requirement into two or more software quality requirements. For example, in a distributed system, fault-tolerance can be divided into fault-tolerant computation and fault-tolerant communication.

Software Architecture Description


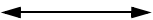
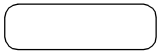
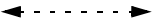


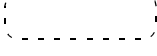
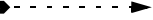
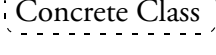



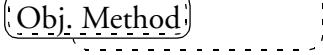
The description of software architectures is an important issue, since very many persons are dependent on the software architecture for their work, e.g. project managers use it for estimating effort and planning resources, software engineers use it as the input to the software design, etc. The inherent problem in most of software development is the abstract nature of a computer program. Unlike products in other engineering fields, e.g. cars, houses, or airplanes, software is non-tangible and has no natural visualization. Currently, the most accurate description of the system is its source code, or rather the compiled code since most compilers have their own ways of compiling, structuring and optimizing the source code. But even then are we dependent on descriptions of the execution platform, e.g. computer architecture, processor and operating systems. Hence the problem with software architecture description is to find a description technique that suites the purposes in software development, i.e. communication, evaluation and implementation. In this section some description techniques will be presented, starting with the most commonly used, boxes and lines.

Boxes and lines

Most architectures meet the first light on a white-board in form of an informal figure consisting of a number of boxes with some text or names and lines to connect the related boxes. The boxes and lines notation is very fast to draw and use on paper and on white-boards. During a work meeting the participants have enough context by following the discussion and

modifications to the figure to understand what it means and how it should be interpreted. Without the context given in such a meeting, the description consisting of only boxes with names and lines to connect them could be interpreted in very many ways and give but a very coarse picture of the architecture. The modules have important properties that are not described at all by simple boxes, e.g. the public interface of the module, control behavior, and the rationale. This is why the boxes and lines description techniques are needed, but not sufficient. After the first descriptions using boxes and lines, the architecture elements need to be specified in more detail, and for this the description technique needs to be more expressive. In (Bass *et al.*, 1998) an extended boxes and lines notation for describing software architecture is presented with a key to the semantic interpretation (see figure 5 for partial description of the notation).

Figure 5.
*Software
Architecture
Notation*

Components		Connectors	
	Process	Uni/Bi-Direc- tional Ctrl Flow	
	Computational Components	Uni/Bi-Direc- tional Data Flow	
	Active Data Component	Data & Control Flow	
	Passive Data Component	Implementation	
	Concrete Class	Aggregation	
	Abstract Class	Inheritance	
	Obj. Method		

The notation includes the concepts of aggregation and inheritance, without a semantic definition. There is a risk of mistaking the inheritance and aggregation concepts proposed in this notation for the semantics of the same words in object-orientation.

Inheritance in software architecture could be interpreted as a module that inherits from another module has the same interface as the super module. In object orientation this is also the case, but the implementation would also be inherited. However, even in object oriented languages the interpretations differ, for example Java and C++ have different inheritance constructs. Java uses the inheritance mechanism for code reuse, whereas the interface construct is used to denote type compliance.

Aggregation also has no definition of its semantic interpretation in software architecture. In object orientation aggregation would be interpreted as an object being composed of other objects, i.e. nested objects. In software architecture, that interpretation would imply nested modules. This, however, seems less probable since the module/component concept, as it is used in industry (Bosch, 1998), is that a module could be fairly large and consists of several classes.

Parameterization together with interfaces play an important role in software architecture. When a service of a module is called by another module, the caller also supplies a reference to one of the modules that should be used to carry out the service request. For example, a database request is sent to the database manager module, with references to the database connection to be used for the query and a reference to the container component where the query results are to be stored or processed. This parameterization could be seen as a kind of module relation, but the notation does not allow unambiguous specification.

Multiple views

Software architecture can be viewed from many different perspectives. All relevant aspects of an software architecture cannot be visualized conveniently in one single type of diagram. The general approach to deal with this problem is to describe software architecture from different *viewpoints* (IEEE 1471), or views using appropriate types of diagrams for each viewpoint.

In (Bass *et al.*, 1998) the viewpoints of the architecture are called architectural structures and every stakeholder is concerned with, at least, one structure. A few of the examples of potential structures are listed here:

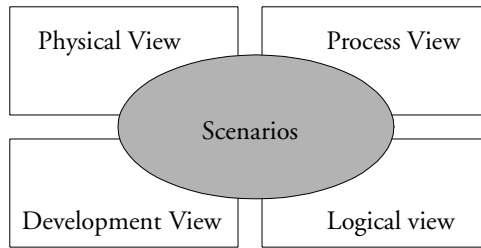
- 1 Module structure, as the basis for work assignments and products. It is mainly used for project planning and resource allocation.
- 2 Conceptual, or logical structure, as the description of partitioning of the system's functional requirements.
- 3 Process structure, or coordination structure, describes the control flow of the system, mainly in terms of processes and threads.
- 4 Physical structure describes the hardware entities that are relevant to the system.
- 5 Uses structure, to show dependencies between modules.
- 6 Calls structure, to show the utilization of other modules' services or procedures.
- 7 Data flow, describes the data flow between modules, i.e. what modules send or accept data from other modules.
- 8 Control flow

Views, or structures, are dependent on elements from each other (Bass *et al.*, 1998). However, traceability between the views is not obvious. In a small system the similarities between views are more than the differences, but as the system grows the more significant differences between views becomes.

The 4+1 View Model - Kruchten

On page 27 the design method aspects of the 4 + 1 View Model (Kruchten, 1995) is presented, but it is primarily intended as a way to describe the software architecture from five different perspectives (Figure 6). Each view addresses concerns of interest to different stakeholders and has an associated description method. Three of the views in the 4 +1 View Model use subsets of UML, the fourth is the structure of the code and the fifth (+1) is a list of scenarios specified in natural language text. The 4+1 View Model was developed to rid the problem of software architecture representation. On each view, the Perry/Wolf definition (discussed on page 20) is applied independently. Each view is described using its own representation, also called *blueprint*.

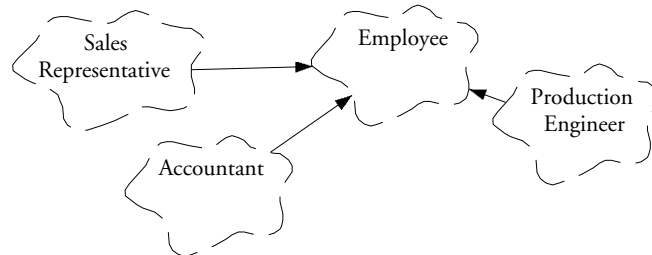
Figure 6.
*Each view address
 specific concerns*



Logical View The logical view denotes the partitions of the functional requirements onto the logical entities in the architecture. The logical view contains a set of key abstractions, taken mainly from the problem domain, expressed as objects and object classes. *If an object’s internal behavior must be defined, use state-transition diagrams or state charts.*

The object-oriented style is recommended for the logical view. The notation used in the logical view is the Booch notation (Booch, 1999). However, the numerous adornments are not very useful at this level of design.

Figure 7.
*Booch notation
 example in Logical
 View*



Process View The process view specifies the concurrency model used in the architecture. In this view, for example, performance, system availability, concurrency, distribution system integrity and fault-tolerance can be analyzed. The process view is described at several levels of abstractions, each addressing an individual concern.

In the process view, the concept of a process is defined as a group of tasks that form an executable unit. Two kinds of tasks exist; major and minor. Major tasks are architectural elements, individually and uniquely addressable. Minor tasks, are locally introduced for implementation reasons, e.g. time-outs, buffering, etc. Processes represent the tactical level of architecture control. Processes can be replicated to deal with performance and availability requirements, etc.

For the process view it is recommended to use an expanded version of the Booch process view. Several styles are useful in the process view, e.g. pipes & filters, client/server (Shaw and Garlan, 1996; Bushmann *et al.*, 1996).

Physical View The elements of the physical view are easily identified in the logical, process and development views and are concerned with the mapping of these elements onto hardware, e.g. networks, processes, tasks and objects. In the physical view, quality requirements like availability, reliability (fault-tolerance), performance (throughput) and scalability can be addressed.

Development View The development view takes into account internal, or, intrinsic properties/requirements like reusability, ease of development, testability, and commonality. This view is the organization of the actual software modules in the software development environment. It is made up of program libraries or subsystems. The subsystems are organized in a hierarchy of layers. It is recommended to define 4-6 layers of subsystems in the development view. A subsystem may only depend on subsystems in the same or lower layers, to minimize dependencies.

The development view supports allocation of requirements and work division among teams, cost evaluation, planning, progress monitoring, reasoning about reuse, portability and security.

The notation used is taken from the Booch method, i.e. modules/subsystems graphs. Module and subsystems diagrams that show import and export relations represent the architecture.

The development view is completely describable *only* after all the other views have been completed, i.e. all the software elements have been identified. However, rules for governing the development view can be stated early.

Scenarios The fifth view (the +1) is the list of scenarios. Scenarios serve as abstractions of the most important requirements on the system. Scenarios play two critical roles, i.e. design driver, and validation/illustration. Scenarios are used to find key abstractions and conceptual entities for the different views, or to validate the architecture against the predicted usage. The scenario view should be made up of a small subset of *important* scenarios. The scenarios should be selected based on criticality and risk.

Each scenario has an associated *script*, i.e. sequence of interactions between objects and between processes (Rubin and Goldberg, 1992). Scripts are used for the validation of the other views and failure to define a script for a scenario discloses an insufficient architecture.

Scenarios are described using a notation similar to the logical view, with the modification of using connectors from the process view to show interactions and dependencies between elements.

Hofmeister et al - View Model

Hofmeister *et al.* (2000) defines, as the basis for their approach, a set of four views to describe software architectures; conceptual view, module view, execution view and code view. The four views have been extracted from the studies of a number of industrial systems. Associated with each of these four views are a specialization of UML to use for notation.

- Conceptual View** The conceptual view is the view closest to the application domain. It is the view least constrained by the software and hardware platform. In this view the system is modelled as a collection of components and connectors that later can be decomposed in the other views. Properties like performance and dependability should be addressed in the conceptual view, but need to be revisited for verification in the execution view. Other properties are less relevant to consider during the conceptual view. For example, portability, is more related to the hardware and software platform, concepts that are addressed in the module view.
- Module View** In the module view the application is modelled using concepts that are present in the programming models and platforms used. So that, for example, it is clear what packages, classes and frameworks, to use and how they should be related. Hence, in this view we bind the architecture to a programming model, whereas in the conceptual view we use a very simple programming model, i.e. that of components and connectors.
- Execution View** The execution of the system is related (as shown in figure 3) to all the other views because it describes the system in terms of the execution related elements, e.g. hardware architecture, processing nodes and operating execution concepts like threads and processes. This view is well suited for addressing properties concerned with system performance, e.g. response

times, throughout, and real-timeliness. In this view replication of parts of the system is addressed, e.g. running several instances of the same program using different processes.

Code View The code view describes the source code organization, the binary files and their relations in different ways. More advanced building approaches and different types of libraries have lead to a need to consciously organize and describe these source and binary files.

The presented views resemble the four views prescribed in the 4+1 View Model, but have significant differences. For example, the execution view corresponds partly to the process view and the physical view, the module view is mostly related to the logical view and the conceptual view does not really have a corresponding view in the 4+1 View Model.

Unified modeling language

The *unified modeling language (UML)* (Booch *et al.*, 1998) has gained an important role in the design of software today. By unifying the main design method and notations the software industry has gained a well thought through design method and notation with a corresponding market of CASE-tools. In UML we find support for classes, abstract classes, relationships, behavior by interaction charts and state machines, grouping in packages, nodes for physical deployment, etc. All this is supported in nine (9) different types of diagrams; class diagram, object diagram, use case diagram, sequence diagram, collaboration diagram, state chart diagram, activity diagram, component diagram and deployment diagram.

In UML we find support for some software architecture concepts, i.e. components, packages, libraries and collaboration. First, the UML allows description of the components in the software architecture on two main levels, either specifying only the name of the component or specifying the interfaces or the classes implementing the components. The notation for a component in UML is shown in figure 8 and a component diagram in figure 9.

Figure 8.
*Component denoted
using UML*

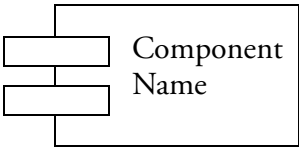
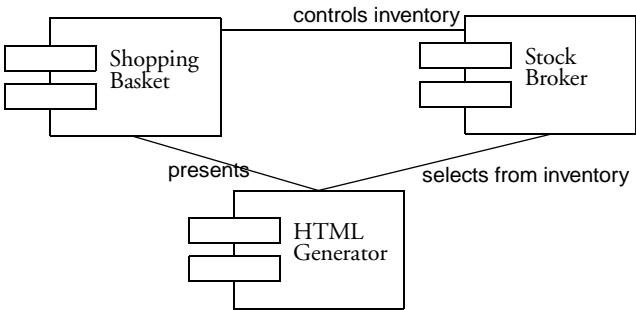


Figure 9.
*An example
component diagram
in UML*



UML provides notational support for describing the deployment of the software components onto physical hardware, i.e. nodes. This corresponds to the physical view in the 4+1 View Model in section . Deployment of the software allows the software engineers to make fewer assumptions when assessing the qualities of the deployed system. Fewer assumptions help in finding a better suited solution for the specific system. The deployment notation, as shown in figure 10, can be extended to show more details about the components deployed on the specific nodes. Nodes can be connected to other nodes using the UML notation, see example of a deployment diagram in figure 11.

Figure 10.
*Node denoted using
UML*

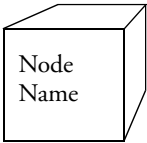
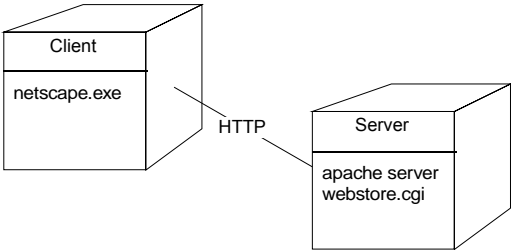
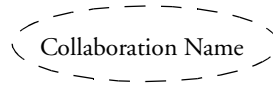


Figure 11.
*An example
deployment diagram
in UML*



Collaborations are sets or societies of classes, interfaces and other elements that collaborate to provide services that beyond the capability of the individual parts. The collaboration has a structural aspect, i.e. the class diagram of the elements involved in the collaboration, and a behavior diagram, i.e. interaction diagrams describing different behavior in different situations. Collaborations also have relationships to other collaborations.

Figure 12.
*Notations for
collaboration in
UML*



Patterns and frameworks are supported in UML by combined usage of packages, components, collaborations and stereotypes.

Architecture description languages

More formal approaches to describing software architectures have emerged in form of *architecture description languages* (ADL). In comparison to requirement specification languages that are more in the problem domain, software architecture description languages are more in the solution domain. Most architecture description languages have both a formal textual syntax and a graphical representation that maps to the textual representation. ADLs should, according to Shaw *et al.* (1995), have; ability to represent components and connectors, abstraction and encapsulation, types and type checking, and an open interface for analysis tools. Luckham *et al.* (1995) prescribe that architecture description languages should have component and communication abstraction, communication integrity, model support for dynamic architectures, causality and time support, and relativity or comparison.

At this point, over ten architecture description languages have been presented, e.g. Rapide (Luckham *et al.*, 1995), and Unicon (Shaw *et al.*, 1995). In Clements (1996) eight ADLs are surveyed and compared on different features, and Shaw *et al.* (1995) propose a framework for comparison of software architecture description languages and comparison of number of existing architecture description languages. Recently, Medvidovic and Taylor (2000) proposed a classification framework for architecture description languages.

We consider a system's software architecture as the first design decisions concerning the system's structure: the decomposition of the system into components, the relationships between these components and the relationship to its environment. These design decisions may have a considerable influence on various qualities of the resulting system, including its performance, reliability and modifiability. However, these qualities are not completely independent from each other, and as already mentioned they may interact; a decision that has a positive effect on one quality might be very negative for another quality. Therefore, trade-offs between qualities are inevitable and need to be made explicit during architectural design. This is especially important because architectural design decisions are generally very hard to change at a later stage. So, we should analyze the effect of these design decisions before it becomes prohibitively expensive to correct them. That is the rationale for performing explicit software architecture analysis.

The goal with software architecture analysis is to learn about the system to be built with the software architecture. This kind of analysis requires mappings between the software architecture and the system to be built. The accuracy of the results from such analyses are very dependent on how ambiguous these mappings are. No such universal mapping exists and that is not because the lack of attempts, e.g. UML, but because the precise mapping is dependent on many factors in the specific case. It is, for example dependent on the platforms, APIs and programming languages chosen in each specific case. The current state of the research and practice make use of what is available and the semantics of the software architecture description are shared by stories, written English text, and usage of other related description techniques, e.g. UML, OMT, or state charts.

The analysis of software architecture for the purpose of learning about the system that is going to be implemented would benefit from having a clear and universally defined semantics of a software architecture description technique.

Architecture analysis must take into account that when the software architecture is designed; detailed design is done on every module in the architecture and the implementation. This is a source of variation in what could be expected from the

implemented system. For example, a brilliant team of software engineers may still be able to do a good job with a poor software architecture. On the other hand, a perfect software architecture may lead to unacceptable results in the hand of a team of inexperienced software engineers that fail to understand the rationale behind the software architecture.

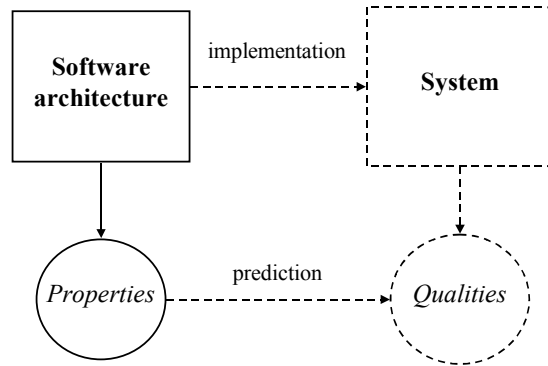
Most engineering disciplines provide techniques and methods that allow one to predict quality attributes of the system being designed even before it has been built. The availability of such techniques is of high importance since without such techniques engineers may construct systems that fail to fulfill their quality requirements.

Prediction techniques and methods can be employed in every stage of the development process. Code metrics, for example, have been investigated as a predictor of the effort of implementing changes in a software system (Li & Henry, 1993). The drawback of this approach is that it can only be applied when a substantial part of the system's code has been written, which is only after a large effort has been put in the development of the system. At that time it often is prohibitively expensive to correct earlier design decisions. Other authors investigate design-level metrics, such as metrics for object-oriented design (Briand *et al.* 1999). Such an approach is useful in the stages of development where (part of) the object-oriented design is finished.

The first steps in meeting the quality requirements for a system are taken in the design of the software architecture. These decisions strongly influence the quality levels that may be achieved in the resulting system. Thus, it is important to verify that the software architecture supports the required quality levels. Such verification requires techniques that allow one to predict a system's qualities based on its software architecture. To do so, we study relevant properties of the artifact that is present, namely the software architecture (indicated by the solid lines in Figure 13). Based on these properties, the analysis then aims at stating predictions about the qualities of a system that has not been implemented yet (indicated by the dotted lines in Figure 13). Because the architecture design does not fully specify and implement the system there are still several design decisions that have to be made after the software architecture design is finished. This fact inflicts a limitation on the predictions we are able to make from the software architecture,

namely that we cannot guarantee that subsequent design decisions will not decrease the level of a quality attribute in the implemented system. These are the principles governing software architecture analysis.

Figure 13.
*Software
architecture analysis*



Within the development process, software architecture analysis can be performed with two main perspectives of the assessors, i.e. internal and external. Typically, the external perspective is taken when an assessment team that is external to the development project carries out the analysis. An external analysis is mainly used at the end of the software architecture design phase as a tool for acceptance testing ('toll-gate approach') or during the design phase as an audit instrument to assess whether the project is on course. The internal perspective is mostly taken when the software architect performs software analysis as an integrated part of the software architecture design process. The selected perspective does not necessarily control the approach taken for the analysis, but rather influences the interpretation of the results and their applicability.

Using scenarios for software architecture analysis

The basic principle of scenario-based analysis was already introduced by David L. Parnas in 1972 (Parnas, 1972). In scenario-based analysis, possible sequences of events are defined and their effect on the system is assessed. Although Parnas did not use the term scenario, such descriptions of a sequence of events are equivalent to scenarios. Recently, the role of scenarios was introduced in software architecture (Kazman *et al.*, 1994; Kazman *et al.*, 1996; Kazman *et al.*, 1998).

In modifiability analysis, scenarios are used to capture future events that require the system under analysis to be adapted. We

will refer to this type of scenarios as *change scenarios*. This use of the term scenario is different from conventional approaches where it is used solely to refer to usage scenarios. Examples of these approaches include object-oriented design (Jacobson *et al.* 1992) with its use cases (scenarios) and scenario-based requirements engineering (Sutcliffe *et al.* 1998).

The main reason for using change scenarios in software architecture analysis of modifiability is that they are usually very concrete, enabling us to make detailed analysis and statements about their impact. At the same time, this can pose a threat to the value of our analysis when our set of change scenarios is not selected with care. The implication of having scenarios that are too specific is that the results of the analysis cannot be generalized to other instances of change.

Scenario-based architecture assessment method

One of the earliest methods for software architecture analysis that uses scenarios was SAAM (Software Architecture Analysis Method) (Kazman *et al.* 1996; Bass *et al.*, 1998). SAAM is used to assess software architectures before the detailed design and implementation has started. It should involve all stakeholders of the architecture. The goal with the assessment is to make sure that all stakeholders' interests will be accounted for in the architecture. SAAM consists of six steps (Bass *et al.*, 1998):

- 1 *Scenario development* is the activity where each stakeholder lists a number of situations, usage situations, or changes, that are relevant for him/her concerning the system.
- 2 The software *architecture description* serves as the input together with the scenarios for the subsequent steps of the method. The description should be in a form that is easily understood by all stakeholders.
- 3 The leader of the SAAM session directs the *classification of scenarios* into direct or indirect scenarios. Direct scenarios means that it is clear that this scenario is no problem to incorporate or execute in the implemented system. Indirect scenarios mean that it is not clear whether a scenario can or cannot be directly incorporated in the architecture. The purpose of the classification is to reduce the number of scenarios that is used as input for the next step in the method.

- 4 The indirect scenarios are now *evaluated individually* to remove any doubts as to whether or not the scenarios are direct or indirect.
- 5 The specified scenarios are mostly related to some extent and sometimes require changes to the same components. This is ok when the scenarios' semantics are related closely, but when semantically very different scenarios require changes to the same components it may be an indication of problems. *Scenario interaction assessment* exposes these components in the architecture.
- 6 In the case when architectures are compared the *overall evaluation* plays an important role. The overall evaluation is the activity of quantifying the results from the assessment. Assigning all scenarios weights of the relative importance does this.

Architecture trade-off analysis method

SAAM has evolved into the Architecture Tradeoff Analysis Method, or ATAM (Kazman *et al.* 1998; Kazman *et al.* 1999; Kazman *et al.* 2000a). The purpose of ATAM is to assess the consequences of architectural decisions in relation to the quality requirements. ATAM uses scenarios for identifying important quality attribute requirements and for testing the architectural decisions' ability to meet these requirements.

The quality requirements are related to architectural decisions, captured in Architecture-Based Architectural Styles (ABASs) to find trade-off points, i.e. properties that affect more than one quality attribute. A number ABASs have been presented, but the coverage cannot be complete. The emphasis of ATAM is not on an in-depth assessment of a single quality attribute, but rather on identifying trade-offs between quality attributes. The assessment is divided into two phases.

These are the steps of the method (Kazman *et al.*, 2000a):

- 1 The assessment team *present the ATAM* to the stakeholders.
- 2 A stakeholder with business perspective *presents the business drivers*. The business drivers are the most important aspects to the business case.
- 3 The architect *presents the architecture* at an appropriate level.

- 4 The architect *identify architectural approaches* and the assessment team records them.
- 5 The stakeholders, together with the assessment team, *generate the quality attribute utility tree*. The purpose is to arrive at a model for prioritizing the requirements.
- 6 The assessment team then *analyzes the architectural approaches*. The idea is to understand what approaches are used for the most important quality requirements.
- 7 The stakeholders *brainstorm and prioritizes scenarios* to be used in the analysis.
- 8 At this point the architect completes the analysis of the architectural approaches by mapping the scenarios to the elements in the descriptions of the architecture.
- 9 Present the results.

Global Analysis

Based on three categories of factors that influence the software architecture, Hofmeister *et al.*, (2000) present the *global analysis method*. It is a major part of the whole software architecture design approach. The name is motivated in these three ways:

- Many of the factors affect the entire system and should be addressed with strategies on how to implement them locally in the components and subsystems.
- The global analysis activity take place throughout the software architecture design. As the design progresses, new knowledge and information reveal issues and concerns previously unknown.
- Factors are considered as a group, rather than single issues. Many factors have dependencies, some of these dependencies may be counteracting.

The main tasks in the global analysis method are; analyze factors and develop strategies. In the description on their method a number of questions that should be asked are presented. Also templates for recording the results of the global analysis is provided.

Analyze Factors

- 1 Identify and describe the factors. Primarily, the factors that have global influence or can change during the development are prioritized. The factors are organized in three cat-

egories; organizational factors, technological factors and product factors. These factors may be explicit from other sources, but if not, identifying them is the first step.

- 2 Characterize their flexibility and changeability. The character of a factor describes what is negotiable about the factor.
- 3 Analyze their impact. The impact of concern is the factors' impact on; other factors, components, system operation and other design decisions. The goal is to express the impact in terms of how particular components are affected.

Develop Strategies

The second main task of global analysis is to address each of the factors with strategies on how to implement them in the different parts of the software architecture.

- 1 Identify issues and influencing factors. The issues is identified in the factor descriptions and their impact analysis results. Issues depend on, for example, the volatility of a factor or the difficulty in satisfying a requirement.
- 2 Develop solutions and specific strategies. Solution strategies should be designed such that the respective factor's influence is kept localized. It is also part of this task to make sure that the developed strategies are consistent.
- 3 Identify related strategies. Strategies may relate to many factors and issues but should only be described once. Issue cards should be used to describe each issue and the strategies to address it.

Architecture discoveries and reviews

At AT&T architecture assessment has been divided into two kinds, *architecture discovery* and *architecture review* (Avritzer and Weyuker, 1998). Architecture discovery is used very early in the project and helps the development group make decisions and balance benefits and risks. The motivation for architecture discoveries is to find and evaluate alternatives and associate risks. The second assessment type, architecture review, is done before any coding begins. The goal is to assure that the architecture is complete and identify potential problems. The best technical people not belonging to the project perform architecture assessments. The review team chairperson and the project's lead architect assign reviewers in cooperation to ensure that the review team incorporates the top competence in *all* areas relevant to the project.

Checklists are employed for software architecture analysis addressing issues related to different qualities, among others modifiability (Maranzano 1993). These lists are used to check whether the development team followed ‘good’ software engineering practices, i.e. practices that are assumed to increase the quality of the system. However, this does not guarantee that the system will be modifiable with respect to changes that are likely to occur in the life cycle of the system. This issue is addressed in scenario-based software architecture analysis methods, which investigate the effect of concrete changes to assess a system’s modifiability.

Conclusions

Software architecture has many definitions. Most definitions share the view that software architecture concerns the division of the software into modules, or components, and the relations between those modules.

Software architecture is put in a business context by the Architectural Business Cycle (Bass *et al.*, 1998) that reveals the feedback from the software architecture and system.

We have presented five different software architecture design methods; use of architecture patterns (Bushman *et al.*, 1996), Recursive design (Shlaer and Mellor, 1997), 4+1 View Model (Kruchten, 1996), architecture design/Global analysis (Hofmeister *et al.*, 2000), and QASAR (Bosch, 2000). The design methods all emphasize the importance of achieving the correct quality characteristics in the implemented system. Two of the presented methods are integrated design, description and analysis methods, *viz.* 4+1 View Model and architecture design /Global Analysis. All these methods address in different ways how to know when the design is done. The 4+1 does not describe any process or method for doing so but prescribes the +1 view to be used. Global Analysis is used as an integrated part of the design, and the design is ready when all issues cards have acceptable strategies. In QASAR the placeholder for an evaluation method is presented and a scenario-based approach is outlined.

Concerning software architecture description we conclude that there is a consensus on the need for multiple viewpoints when describing software architecture, manifested in the IEEE 1471

recommended standard. Bass *et al.* (1998) enumerates a long list of viewpoints that can be used. In the 4+1 View Model, Kruchten (1996) defines four views for describing the architecture and a fifth to drive the design and verification. Hofmeister *et al.* (2000) also defines four views, although they have significant differences from the views of the 4+1 View Model. For example, the logical view in the 4+1 View Model and the conceptual view (Hofmeister *et al.*, 2000) may seem similar at a first glance, but the conceptual view is not bound to any programming paradigm as opposed to the logical view. Those concerns are addressed later in the module view.

Software architecture analysis can be performed in many different ways and for different reasons with different goals. In this chapter we have presented four different ways to analyze software architecture; SAAM (Kazman *et al.*, 1996; Bass *et al.*, 1998), ATAM (Kazman *et al.*, 1998; Kazman *et al.*, 1999; Kazman *et al.*, 2000), Global Analysis (Hofmeister *et al.*, 2000), and Architecture discoveries and reviews (Avritzer and Weyuker, 1998; Maranzano, 1993). SAAM and ATAM make use of scenarios to drive the analysis, whereas Global Analysis use factors, issues and strategies. Architecture discoveries and reviews are based on checklist. SAAM, ATAM and Architecture Reviews and Discoveries focuses on evaluation of the finished version of the software architecture to determine whether it is fit for detailed design and implementation. Global analysis, on the other hand is part of the design process.

It is our conclusion that there is a need for analysis methods that can serve different goals both integrated with the design process and as a verification step at the end of design.

CHAPTER 2: **Modifiability Analysis**

The world around software systems is constantly changing. Software that used to function properly turn incompatible because its environment change. For this reason most software systems need to be modified many times after their first release. Another reason for modifying software is when software products are updated and improved to keep the competitive advantage against other products on the market. The result is software products that evolve from release 1, to release 2, to release 3, and so forth. Several examples of this exists, for instance, Adobe Photoshop that is currently in its sixth major release. In developing a software product today, it is accepted and expected that there will be subsequent releases of the product. The software life-cycle of a product is continuous development that only ends when the product is obsolete.

The changes that are implemented from release to release include anticipated changes, e.g. following the market plan, and unanticipated ones, e.g. provoked by sudden and unexpected changes from different sources. Additionally, the bug fixes for the previous releases are usually incorporated into the code baseline. Based on our understanding about the future of the software system, we can choose the design alternatives that supports future modifications better over otherwise equivalent alternatives.

The goal is to increase the productivity in the subsequent releases by choosing the most appropriate design solutions from the start. Consequently, stakeholders are generally interested in a system designed such that future changes will be relatively easy to implement, and thus increase the maintenance productivity for implementing these changes. Typical questions that stakeholders pose during the early design stages, i.e. software architecture design, include:

- Which of the available architecture designs requires the lowest cost to maintain, i.e. yields the highest productivity to accommodate future changes?
- How much effort will be needed to develop coming releases of the system?
- Where are the trouble spots in the system with respect to accommodating changes?
- Is there any other construction that is significantly easier to modify than the present one?

These questions concern a system's ability to be modified.

Modifiability

In the software engineering literature a large number of definitions of qualities exist that are related to this ability. A very early classification of qualities (McCall *et al.* 1977) includes the following definitions:

Maintainability is the effort required to locate and fix an error in an operational program.

Flexibility is the effort required to modify an operational program.

A more recent classification of qualities is given in the ISO 9126 standard. This standard includes the following definition, related to modifying systems:

Maintainability is the capability of the software product to be modified. Modifications may include corrections, improvements or adaptations of the software to changes in environment, and in requirements and functional specification.

Although different in wording, their semantics do not differ so much. For our purpose the scope of these definitions is too broad. They capture a number of rationales for system modifications in a single definition, i.e. bugs, changes in the environment, changes in the requirements and changes in the functional specification. We consider only the latter three of these change categories. The reason for excluding bugs is a basic assumption that the remainder of the process will adhere to the specifications made and that faults are only introduced unintentionally. One could maybe find design heuristics that

help expose solutions that are likely to render more faults than others. However, that is a very different approach than the one taken for this method. Consequently, we arrive at the following definition of modifiability, which we use in the remainder of the thesis:

The modifiability of a software system is the ease with which it can be modified to changes in the environment, requirements or functional specification.

This definition demonstrates the essential difference between maintainability and modifiability, namely that maintainability is concerned with the correction of bugs whereas modifiability is not.

Analyzing Modifiability

Quality requirements such as performance and maintainability are, in our experience, generally specified rather weakly in industrial requirement specifications. In some of our cooperation projects with industry, e.g. (Bengtsson and Bosch, 1999), the initial requirement specification contained statements such as “The maintainability of the system should be as good as possible” and “The performance should be satisfactory for an average user”. Such subjective statements, although well intended, are useless for the evaluation of software architectures.

To predict the quality attributes of a system from the software architecture, it is required that one specify quality requirements in sufficient detail. One common characteristic for quality requirements is that stating a required level without an associated context is meaningless. For instance, also statements such as “Performance = 20 transaction per second” or “The system should be easy to maintain” are meaningless for architecture evaluation.

One common denominator of most quality attribute specification techniques is that some form of *scenario profile* is used as part of the specification. A scenario profile is a set of scenarios (see “Change Scenario Profiles” on page 74). The profile used most often in object-oriented software development is the *usage profile*, i.e. a set of usage scenarios that describe typical uses for the system. The usage profile can be used as the basis for specifying a number of, primarily

operational, quality requirements, such as performance and reliability. However, for other quality attributes, other profiles can be used. For example, for specifying safety we have used *hazard scenarios* and for modifiability we have used *change scenarios* (See “Change Scenarios” on page 73).

Architecture-Level Modifiability Analysis Method (ALMA)

In the year 2000, myself, Professor Jan Bosch, Nico Lassing and Professor Hans van Vliet developed a method for scenario-based architecture-level modifiability analysis method (ALMA). ALMA has the following characteristics:

- Focus on modifiability.
- Distinguish multiple analysis goals.
- Make important assumptions explicit.
- Provide repeatable techniques for performing the steps.

ALMA consists of five main steps, i.e. goal selection, software architecture description, scenario elicitation, scenario evaluation and interpretation.

Background

When discussing our independent modifiability analysis methods and experiences with using them, we found that modifiability analysis generally has one of three goals. It is either one of these; prediction of future modification costs, identification of system inflexibility, or comparison of two or more alternative architectures. Depending on the goal, the method is adapted by using different techniques in some of the main steps.

Further, our discussions reveals that the domains in which we had applied our previous methods influenced the assumptions made. The different goals and domains revealed differences in required information, different approaches to scenario elicitation (including the notion of a ‘good scenario’). In unifying our methods, these previously implicit assumptions became explicit decisions. The result is that the new method has increased repeatability and an improved basis for interpreting the findings.

Process

ALMA has a structure consisting of the following five steps:

- 1 Set goal: determine the aim of the analysis
- 2 Describe software architecture: give a description of the relevant parts of the software architecture
- 3 Elicit scenarios: find the set of relevant scenarios
- 4 Evaluate scenarios: determine the effect of the scenarios
- 5 Interpret the results: draw conclusions from the analysis results

When performing an analysis, the separation between the tasks is not very strict. It is often necessary to iterate over various steps. For instance, when performing scenario evaluation, a more detailed description of the software architecture may be required or new scenarios may come up. Nevertheless, in the next subsections we will present the steps as if they are performed in strict sequence.

Depending on the situation, we select a combination of techniques to be used in the various steps. The combination of techniques cannot be chosen at random, certain relationships between techniques exist. These are discussed in “Relating the steps” on page 61.

Setting the goal

The first activity in ALMA is concerned with the goal of the analysis. It is not an activity in the same sense as the remainder of the steps, rather it is a decision to be made. By making this decision a step, or decision point, in the beginning of the process we emphasize the importance of a clear goal for the analysis. Architecture-level modifiability analysis can target the following goals:

- Maintenance prediction: estimate the effort that is required to modify the system to accommodate future changes
- Risk assessment: identify the types of changes for which the software architecture is inflexible
- Software architecture selection: compare two candidate software architectures and select the optimal candidate

Architecture description

The second step, architecture description, concerns the information about the software architecture that is needed to perform the analysis. Generally speaking, modifiability analysis requires architectural information that allows the analyst to evaluate the scenarios. Scenario evaluation comprise two tasks; analysis of the scenario impact and estimation of the impact's extent.

Architecture-level impact analysis is concerned with identifying the architectural elements affected by a change scenario. This includes, obviously, the components that are affected directly, but also the indirect effects of changes on other parts of the architecture. The effect of the scenario is expressed using some measurement, depending on the goal of the modifiability analysis.

Available Information

During the course of architecture design the software architecture evolves. It is extended gradually with architectural design decisions. As a consequence, the amount of information available about the software architecture is dependent on the point in time at which the analysis is performed.

Initially, we base the analysis on the information that is available. If the information proves to be insufficient to determine the effect of the scenarios found, there are two options. Either we can decide that it is not possible to determine the effect of the scenario precisely and postpone the analysis, or, the architect that assists the scenario evaluation may provide the missing information. When providing information during the analysis, additional architectural decisions may be made that should be documented. In the such cases architecture description is not just an observation activity, but it is an aid in software architecture design as well.

View Models

It is not within the scope of this thesis to define nor develop software architecture description techniques, or notations. Instead it is valuable to keep the analysis method independent from the description techniques and notations. Practitioners use different description techniques at different maturity levels, and as long as the minimal information is captured, we should be able to use the analysis method. In architecture-level impact analysis the following information about the software architecture need to be communicated by the description:

- The decomposition of the system in components
- The relationships between components
- The relationships to the system's environment

Relationships between components and between the system and its environment come in different forms and are often defined implicitly. They may come from functional dependencies, i.e. a component uses and depends on the interface of another component, synchronization, data flow or some other kind of dependency. Information about these dependencies is valuable in impact analysis; dependencies determine whether modifications to a component require adaptations to other components as well, i.e. causing *ripple effects*.

Notation To capture the information of the viewpoints, different techniques are available. On the one end, there are informal techniques using boxes and lines with little or no explicit semantics. On the other extreme, we have more detailed and formal techniques, such as architecture description languages (ADL), with strict explicit semantics. Currently, the trend is to use notation techniques from UML, e.g. as in Hofmeister *et al.* (1999), which could be viewed as located in the middle of the scale.

Scenario Elicitation

Change scenario elicitation is the process of finding and selecting the change scenarios that are to be used in the evaluation step of the analysis. Eliciting change scenarios involves such activities as identifying stakeholders to interview, properly documenting the scenarios that result from those interviews, etc. Scenario elicitation is described in detail in chapter 4 “Scenario Elicitation” on page 73 and in chapter 5 “An Experiment on Scenario Elicitation” on page 83.

Change scenario evaluation

The fourth step, change scenario evaluation, is concerned with the effect of the change scenarios on the architecture. In this step, the analyst cooperates with, for example, the architects and designers. Together they determine the impact of the change scenarios and express the results in a way suitable for the goal of our analysis. This is architecture-level impact analysis.

To the best of our knowledge no architecture-level impact analysis method has been published yet. A number of authors (Bohner 1991; Kung *et al.* 1994) have discussed impact analysis methods focused on source code. Turver and Munro (1994) propose an early impact analysis method based on the documentation, the themes within the documents, and a graph theory model. However, these methods are not useable in our type of analysis, where we may have nothing more than the software architecture of the system. In general, impact analysis consists of the following steps:

- 1 Identify the affected components
- 2 Determine the effect on the components
- 3 Determine ripple effects

The first step is to determine the components that need to be modified to implement the change scenario.

The second step is concerned with identifying the functions of the components that are affected by the changes. This impact can be determined by studying the available documentation for the component such as, for instance, a specification of its interface. Changes may propagate over system boundaries; changes in the environment may impact the system or changes to the system may affect the environment. So, we also need to consider the systems in the environment and their interfaces. Depending on the detail and amount of information we have available (step 2), we may have to make assumptions and document these as issues.

The third step is to determine the ripple effects of the identified modifications. The occurrence of ripple effects is a recursive phenomenon in that each ripple may have additional ripple effects. Because not all information is available at the architecture level, we have to make assumptions about the occurrence of ripple effects. At one extreme, we may assume that there are no ripple effects, i.e. that modifications to a component never require dependent components to be adapted. This is an optimistic assumption and it is not very realistic. At the other extreme, we may assume that each component related to the affected component requires changes. This is an overly pessimistic assumption and it will exaggerate the effect of a change scenario. In practice, we rely on the

architects and designers to determine whether adaptations to a component have ripple effects on other components.

Evidence on Accuracy

Empirical results on source code impact analysis indicate that software engineers, when doing impact analysis predict only half of the necessary changes (Lindvall & Sandahl 1998). Lindvall and Runesson (1998) report the results from a study of the changes made to the source code between two releases and whether these changes are visible in the object-oriented design. Their conclusion is that about 40% of the changes are visible in the object-oriented design. Some 80% of the changes would become visible, if not only the object-oriented design is considered but also the method bodies. These results suggest that we should expect impact analysis at the architectural level to be less complete because less design details are known and not all required changes will be detected. This issue should be taken into account in the interpretation of the results.

Expressing Impact

After we have determined the impact of the scenarios, we must express the results in some way. We can choose to express these qualitatively, e.g. a description of the changes that are needed for each change scenario in the set. On the other hand, we can also choose to express the results using quantitative measures. For instance, we can give a ranking between the effects of scenarios, e.g. a five level scale (+ +, +, +/-, -, - -) for the effect of a scenario. This allows us to compare the effect of scenarios. We can also make absolute statements about the effort required for a scenario, such as an estimate of the size of the required modification. This can be done using size metrics like estimated lines of code (LOC), or estimated function points (Albrecht, 1979; Albrecht and Gaffney, 1983). The selected technique should support the goal that we have set for the analysis.

Interpretation

When we have finished the change scenario evaluation, we need to interpret the results to draw our conclusions concerning the software architecture. The interpretation of the results depends entirely on the goal of the analysis and the system requirements.

Relating the steps

The techniques used for the various steps should not be chosen at random. Based on the goal of the analysis, we select the appropriate approach to scenario elicitation. As mentioned,

different situations require different approaches to elicitation. For instance, when the goal is to estimate the maintenance costs for a system, we are interested in a different set of scenarios than when the goal is to assess the risks. In the first case, the method requires a set of scenarios that is representative for the actual events that will occur in the life cycle of the system and in the latter case the method requires scenarios that are associated with particularly troublesome changes.

Similar goes for the scenario evaluation technique. Based on the analysis goal, we select a technique for evaluating and expressing the effect of the scenario. For example, if the goal of the analysis is to compare two or more candidate software architectures, we need to express the impact of the scenario using some kind of ordinal scale. This scale should indicate a ranking between the different candidates, i.e. which software architecture supports the changes best.

The evaluation technique then determines what techniques should be used for the description of the software architecture. For instance, if we want to express the impact of the scenario using the number of lines of code that is affected, the description of the software architecture should include a size estimate for each component.

Conclusions

In this chapter we present ALMA, a method for software architecture analysis of modifiability based on scenarios. This method consists of five major steps: (1) set goal, (2) describe the software architecture, (3) elicit change scenarios, (4) evaluate change scenarios and (5) interpret the results.

ALMA distinguishes that one goal can be pursued in software architecture modifiability analysis at a time. The goal pursued restrains the combination of techniques to be used in the subsequent steps. We have identified three main goals: maintenance prediction, risk assessment and software architecture comparison. Maintenance prediction is concerned with predicting the effort that is required for adapting the system to changes that will occur in the system's life cycle. In risk assessment we aim to expose the changes for which the software architecture is inflexible.

After the goal of the analysis is set, we acquire a description of the software architecture. This description will be used in the evaluation of the scenarios. It should contain sufficient detail to perform an impact analysis for each of the scenarios.

The next step is to elicit a representative set of change scenarios, which is to be discussed in detail in the following chapters.

The following step of an analysis is to evaluate the effect of the set of change scenarios. To do so, we perform impact analysis for each of the scenarios in the set. The way the results of this step are expressed is dependent on the goal of the analysis: for each goal different information is required. The final step is then to interpret these results and to draw conclusions about the software architecture. Obviously, this step is also influenced by the goal of the analysis: the goal of the analysis determines the type of conclusions that we want to draw.

CHAPTER 3: Modifiability Prediction Model

When faced with the results of the analysis the architect has to come to a decision on the action to take. If the results are satisfying the action is to proceed. If the results are not satisfying there are two options. Either to improve the architecture in some way, or to proceed because it is not worth the effort to make the improvement. The real question is, of course, how the architect arrives at the conclusion from the analysis results. In our experience, a measure or estimated measure, even if it is well defined, does not provide enough information to form a well informed decision around. In those cases, it is left to the architect to find the frame of reference, i.e. answering the questions; what is reasonable to achieve, what is the worst case, and how far are the results from the best case?

In this chapter we describe a model and technique to reach an estimate, or prediction, of the modification effort for changes of a system. We also present techniques to produce estimates of the best and worst case. These additional estimates, is then used as the frame of reference to make a decision.

Scenario-based Modifiability Prediction

A software system is developed according to a requirement specification, i.e. $RS = \{r_1, \dots, r_p\}$ where r_i is a atomic functional requirement or a quality requirement. Based on this requirement specification, the software architect or architecture team designs a software architecture consisting of a set of components and relations between these components, i.e. $SA = \{C, R\}$. The set of components is $C = \{c_1, \dots, c_q\}$ and each component $c_i = \{I, cs, rt\}$ has an interface I , a code size cs and a requirement trace rt . The set of relations is $R = \{r_1, \dots, r_p\}$, where

each relation is $r_i = \{c_{source}, c_{dest}, type\}$. Thus, relations are directed and one-to-one.

To assess the modifiability of a software architecture, we make use of a maintenance profile consisting of a set of change scenarios, $MP = \{cs_1, ..., cs_j\}$, where each change scenario defines a set of changed and added requirements, $cs_i = \{r_{c,p}, ..., r_{c,p}\}$. $C(MP)$ is the cardinality of the scenario set, i.e. the number of scenarios in the set.

Modification Types

The prediction builds on a model of the modifications' relations to other activities and the total effort. The model can be made very detailed, or more coarse. For example, the productivity can be calculated for the whole release cycle as a relation to the modification size, or it can be calculated separately for different activities in the release cycle. The problem is that adding more detail to that model does not necessarily improve the prediction accuracy, only the effort to make the prediction. In our method we assume a coarse model with three types of modification activities and one overall productivity measure related to lines of code. The activities we assume in the model are:

- 1 adding new components,
- 2 adding new plug-ins to existing components, and
- 3 changing existing component code.

Each of these activities has an associated productivity measure, i.e. P_{nc} , P_p and P_{cc} , respectively. Based on earlier research, e.g. (Henry and Cain, 1997) and (Maxwell *et al.*, 1996), and our own experience the productivity when developing new components for an existing system is roughly an order of magnitude higher than changing the code of existing components.

Once the maintenance profile is available, we can perform impact analysis, i.e. analyze the changes to the software architecture and its components required to incorporate the change scenarios. The result of the impact analysis process is a set of impact analyses, one for each change scenario, i.e.

$IA = \{ia_1, ..., ia_u\}$, where ia_1 is defined as

$ia_i = \{CC_i, NP_i, NC_i, R_i\}$. Here, CC_i is the, possibly empty, set of components that need to be changed for incorporating the change scenario, including a size estimate for the required

change in lines of code, $CC_i = \{\{c_p, s_j\}, \dots\}$. NP_i and NC_i contain similar information for new plug-ins and new components, respectively. R_i contains the new, changed and removed relations required for incorporating the change scenario. CT is the expected number of changes, i.e. estimated change traffic, for the period we want to predict.

As a final step, once the impact analysis has been performed, it is possible to determine the maintenance effort required for the maintenance profile. This can be calculated by summing up the efforts required for each change scenario. Below, the equation is presented:

Equation 1.

Maintenance effort,
(E_M).

$$E_M = \frac{\sum_{AI} \left(\left(\sum_{CC_i} s_j \right) \cdot P_{cc} + \left(\sum_{NP_i} s_j \right) \cdot P_p + \left(\sum_{NC_i} s_j \right) \cdot P_{nc} \right)}{C(MP)} \cdot CT$$

Best and Worst Case Modifiability

Once a quantitative assessment of modifiability has been performed, the next question a software architect typically will ask is: “Does this mean that the modifiability of my architecture is good, reasonable or bad?”. Therefore, it is important to be able to compare this to the maximal modifiability for the scenario profile and the domain.

Best Case

Consider two software architecture alternatives. The first one is based on all kinds of state-of-the-art design techniques to reduce the effort required to make all kinds of modifications in general. The second one is less sophisticated and only make use of a few design techniques to accommodate a limited set of modifications. Which one is the best alternative when it concerns modifiability? Concerning the best alternative in the question above, the answer is; it depends on the modifications we intend to make and which alternative lets us implement them the easiest. The answer requires some explanation. To start with, modifiability is only interesting to the stakeholder that intends to modify the software. Also, the stakeholder is primarily interested in easily making the modifications he or she intends or has to make. Hence, there is no need to accommodate all plausible changes if nobody is interested in

implementing them. So, when we have a list of the relevant modifications, we may find out which of the above alternatives is the better *in our case*.

The worst and best case analysis is based on two assumptions:

Differences in Productivity

First, we assume that the main difference in maintenance effort between systems is due to the difference in productivity between different maintenance activities, i.e. $P_{nc} > P_p \gg P_{cc}$. For example, productivity being (>6x) higher when writing new code (>250 LOC/person-month in C (Maxwell *et al.*, 1996)) compared to modifying legacy code (1,7 LOC/workday ~ <40 LOC/person-month (Henry and Cain, 1997)). Note that these productivity metrics include more activities than simply writing the source statements.

Modification Size Invariance

Second, we assume that the amount of code to be developed or changed for a change scenario is relatively constant in size, independent of the software architecture. In other words, when using the same language, the same programmers and the same domain, modifications require pretty much the same volume of code to be implemented. This means that the impact analysis results from one architecture, i.e. modification size estimates are, in total, roughly the same for any other architecture. This is fundamental to this approach, since it allows us to calculate the results from a hypothetical architecture without manifesting it in a design. If it would be possible to manifest the best possible architecture, the method of doing so should be the design method in the first place.

Based on these assumptions we can formulate the condition for being best case software architecture in terms of modifiability:

The best software architecture alternative is one that allows us to implement the modifications we want at our highest productivity.

It is clear that the best software architecture would allow for the incorporation of all change scenarios by adding new components or, in the case of smaller change scenarios, by adding new plug-ins for existing components.

To assess best and worst case modifiability, we make use of the maintenance profile, *MP*, and the results of the impact analysis, *IA*. The approach to calculate the required effort is straight forwards assuming that all necessary changes can be

implemented as part of a new component. Below, the equation calculating this is presented:

Equation 2.

*Best-case
modifiability effort,
(E_M).*

$$E_M = \frac{\sum_{IA} \left(\left(\sum_{CC_i} s_j \right) + \left(\sum_{NP_i} s_j \right) + \left(\sum_{NC_i} s_j \right) \right) \cdot P_{nc}}{C(MP)} \cdot CT$$

Although this indeed calculates a lower boundary for the required maintenance effort, two issues remain unresolved. First, especially for change scenarios requiring a relatively small change in terms of code size, it may not be reasonable to assume that these are mapped to components. Such change scenarios should be mapped to new plug-ins and the associated productivity metrics should be used. The second issue is a more subtle one. The straightforward model assumes that it is possible that for a maintenance profile, all change scenarios can be mapped to either a new component or a new plug-in. The question is, however, whether it is possible to define a software architecture that supports this. Especially in the case where different change scenarios address the same requirements, it is unreasonable to assume that these change scenarios can be implemented independently, i.e. the changes scenarios interact.

Based on the discussion above, we refine our technique for the assessment of optimal modifiability as follows. First, we categorize change scenarios as independent or interacting. A change scenario is independent if it affects existing requirements in the requirement specification that are not affected by another change scenario that has a lower index number in the maintenance profile than the current scenario. This results in the situation that one change scenario can affect an existing requirement by factoring it out as a separate component or plug-in, but all subsequent change scenarios affecting the same requirement are considered to be interacting.

An interacting change scenario affects one or more requirements in the requirement specification that are also affected by change scenarios with a lower index number. The effort required for this type of change scenarios is calculated based on the ratio of interacting and independent requirements in the set affected by the change scenario. Thus, if, for instance, two of five existing requirements for a change scenario are

interacting, then 40% of the total change size (in, e.g., lines of code) is calculated using the P_{cc} productivity metric. The remainder is calculated by either using the P_{nc} or the P_p productivity metric.

If the total change size of the independent part of the change scenario is less than the code size of the smallest architectural component and/or less than half of the average code size of architectural components, then the P_p productivity metric will be used to calculate the effort. Otherwise, the P_{nc} productivity metric is used.

Thus, for each type of change scenario, the required effort can now be calculated. In equation 3, the alternatives are presented. The term ‘ratio’ refers to the ratio between interacting and independent scenarios. The total maintenance effort can be calculated by summing up the required effort for each change scenario.

Equation 3.

*Realistic Best-case
modifiability effort,
(E_M).*

$$E_M = \sum_{IA} \left(\left(\sum_{CC_i} s_j \right) + \left(\sum_{NP_i} s_j \right) + \left(\sum_{NC_i} s_j \right) \right) \cdot \left(\begin{array}{ll} P_{nc} & \text{if independent \& large scenario} \\ P_p & \text{if independent \& small scenario} \end{array} \right) \cdot \frac{CT}{C(MP)}$$

$$\cdot \left(\begin{array}{ll} (P_{cc} \cdot \text{ratio} + P_{nc} \cdot (1 - \text{ratio})) & \text{if interacting \& large scenario} \\ (P_{cc} \cdot \text{ratio} + P_p \cdot (1 - \text{ratio})) & \text{if interacting \& small scenario} \end{array} \right)$$

Worst case

The worst-case maintainability of a software architecture is where each change scenario has to be implemented by changing existing code, i.e. at the lowest productivity. Whereas the optimal maintainability required a more detailed discussion and alteration of the model, this is not the situation for worst-case maintainability. For instance, most experienced software engineers have experiences with totally eroded systems where each new requirement required changing the source code of the system in multiple locations. The worst-case maintainability can be calculated by summing up the effort required in the worst-case for each change scenario.

Equation 4.

*Worst-case
modifiability effort,
(E_M).*

$$E_M = \frac{\sum_I \left(\sum_{CC_i} \text{size}_{i_j} + \sum_{NP_i} \text{size}_{i_j} + \sum_{NC_i} \text{size}_{i_j} \right) \cdot P_{cc}}{CMP} \cdot CT$$

Conclusions

In this chapter we presented a scenario-based technique for predicting the modifiability effort of a software architecture. The technique employs a scenario profile, used for performing impact analysis and calculates modifiability effort based on this. We also address a problem common to all analysis and prediction methods, namely providing a frame of reference for interpreting and making decisions based on the results.

In addition to the prediction method, the software architect is typically interested in the best- and worst-case modifiability level in order to understand how close the actual level is to the optimal level. Based on two additional assumptions to the prediction model; the difference in effort is because the variations of productivity rather than volume, and that between candidate architectures the modification volume for the same modification are roughly the same, we have presented a way to calculate the modifiability effort of a hypothetical best and worst case. The technique determines the required maintenance effort assuming that all new and changed requirements can be implemented by adding new components or plug-ins to the system.

The limitation is that the method does not manifest the best-case architecture to the analyst.

CHAPTER 4: Scenario Elicitation

Change scenarios drive the modifiability analysis. They describe the modification to be made as well as its context. The scenarios directly affect the relevance of the analysis results. Analyzing irrelevant change scenarios will also render results that are irrelevant. Scenarios have many possible origins. The most important sources of change scenarios are stakeholders that control the modifications of the software in subsequent releases, e.g. product managers. The remainder of this chapter discusses scenarios, scenario profiles and methods for collecting scenario profiles.

Change Scenarios

A change scenario describes changes to be implemented in the software. For example, a change scenario could be:

“Due to changed safety regulations, a temperature alarm must be added to the dialysis fluid module in the dialysis machine”.

There is an important difference between our use of the term ‘scenarios’ compared to Object-Oriented design methods where the term generally refers to use-case scenarios, i.e. scenarios that describe system behavior. Instead, we use the term scenarios to describe an action, or sequence of actions, that might occur in relation to the system. So, a change scenario describes a certain modification task or change that is to be implemented.

A ‘Good’ Scenario

Well specified scenarios for analysis purposes comprise the description of the sequence of events, the triggering event and the desired end state. For well-defined change scenarios this means a short description of the potential modification including the context of the change, why it is needed. The

desired end-state is always a new correctly functioning release of the system. Scenarios should always describe a concrete situation and avoid describing a general type of modification. There are two motivations for this. First, keeping the scenario very specific and detailed allows the analyst to make a more precise evaluation. Second, the scenario is already representative for a whole class of changes, i.e. its *equivalence class*.

Equivalence Classes

The number of possible changes to a system is potentially infinite. Instead of enumerating all changes we partition the space of possible change scenarios into equivalence classes. Each equivalence class consists of all the possible changes that have the same or very similar impact. Then we can treat one scenario from such an equivalence class as a representative instance of the whole class. The result is that we do not need to consider all actual scenarios. Also, we may include scenarios describing modifications already made, but only if we expect more changes from that same equivalence class. The benefit of doing so, is that it is then possible to use the records from the implementation of that previous change in the analysis.

Change Scenario Profiles

A scenario profile is a set of scenarios that together form the context and semantics of software quality factors, e.g. modifiability or safety, for a particular system.

A change scenario profile is a set of change scenarios that form the context for the modifiability requirement posed on a system. Given a certain change scenario profile and a certain estimated change traffic, i.e. estimated number of modification needed, per year, the effort for making the modifications should be less than a certain effort. For example, given the change profile in Table 2 on page 75 and 20 changes per year, the modification effort should be less than 10 kh.

Scenarios may be assigned additional properties, such as an associated weight, priority, or probability of occurrence within a certain time. To describe, for example, the modifiability requirement for a system, we list a number of scenarios that each describe a possible and likely change to the system. An example of a software change scenario profile for the software of a haemo dialysis machine is presented in Table 2. Scenario

profiles represent one way to document and leverage from the experts knowledge about the system.

Table 2: Example of change scenario profile for haemo dialysis

Category	Scenario Description
Market Driven	S1 Change measurement units from Celsius to Fahrenheit for temperature in a treatment.
Hardware	S2 Add second concentrate pump and conductivity sensor.
Safety	S3 Add alarm for reversed flow through membrane.
Hardware	S4 Replace duty-cycle controlled heater with digitally inter- faced heater using percent of full effect.
Medical Advances	S5 Modify treatment from linear weight loss curve over time to inverse logarithmic.
Med. Adv.	S6 Change alarm from fixed flow limits to follow treat- ment.
Med. Adv.	S7 Add sensor and alarm for patient blood pressure
Hardware	S8 Replace blood pumps using revolutions per minute with pumps using actual flow rate (ml/s).
Com. and I/O	S9 Add function for uploading treatment data to patient's digital journal.
Algorithm Change	S10 Change controlling algorithm for concentration of dialysis fluid from PI to PID.

Profile Context

A scenario profile can be defined in one of two basic contexts; the ‘greenfield’ and the experienced context. When a scenario profile is defined in the ‘greenfield’, i.e. an organization uses the technique for the first time on a new system with no historical data available about similar systems. In this case we are fully dependent on the experience, skill and creativeness of the individuals defining the profile. The resulting scenario profile is the only input to the architecture assessment. The lack of alternative data sources in this case and the lack of knowledge about the representativeness of scenario profiles defined by individuals and groups allows for no verification of the scenario profiles produced. The implication of this is that variations between scenario profiles may be caused by instability in the method of creating the scenario profile. That would impact on the assessment method and make the results vary. It is the goal

that the assessment method will produce the same results if the assessment is repeated by (an)other person(s). It is therefore crucial to understand how and if the profiles vary from the three different ways of producing them, i.e. one person, an unprepared group, or a prepared group.

In the second situation, there is either an earlier release of the system or historical data of similar systems available. Since, in this case, empirical data can be collected, we can use this data as an additional input for the next prediction and thus get an more accurate result. However, even when historical data is available to be used as a reference point, it is important that the persons synthesizing the profile also incorporate the new changes that are different from similar systems or the previous release of the system. The problem might otherwise be that, using only the historical data, one predicts the past.

Elicitation Approaches

Change scenario elicitation is the process of finding and selecting the change scenarios that are to be used in the evaluation step of the analysis. Eliciting change scenarios involves such activities as identifying stakeholders to interview, properly documenting the scenarios that result from those interviews, etc. The elicitation process is not specific to change scenarios and could very well be applied for other quality attribute scenarios, but we describe it here for change scenarios, since that is the scope of this thesis. Further, some of these activities are similar to other software development activities. For example, in requirements engineering scenarios are elicited from stakeholders and documented for later use as well (Sutcliffe *et al.*, 1998).

Scenario profile sampling strategies

There are three types of profiles for quality attributes depending on criteria for selecting the scenarios into the profile, i.e. complete set of changes, a sample of changes, and planned changes. The selection criteria influence the representativeness of the scenario profile, since in essence it is a kind of population sampling strategy and the same consequences apply.

When defining a complete profile for a quality attribute, the software engineer defines *all* relevant scenarios as part of the profile. For example, a usage profile for a relatively small system may include all possible scenarios for using the system. Based on this complete scenario set, the software engineer is able to analyze the architecture for the studied quality attribute that, in a way, is complete since all possible cases are included. For most systems, it is impossible to define all possible change scenarios.

The alternative to complete profiles is *selected scenario profiles*. Selected scenario profiles contain a representative subset of the relevant modification, or change scenarios. Assuming the selection of change scenarios has been done carefully, one can assume that the profile represents an accurate representation of the scenario population.

The third criteria is to select those changes that are planned to be made to the software, e.g. company roadmaps for evolving software assets. The scenario profile will then represent the planned changes. Any deviations from those plans are not addressed based on the change scenario profile.

Finding all change scenarios is impossible, which leaves the selected and the planned as strategies to choose from. The planned is strictly limited in how well it will represent future changes, other than those part of the equivalence classes included in the profile. The selected strategy is closest to random sampling, but is not really random. That is a problem, because we cannot guarantee that the scenario profile is a valid representation of the whole population of possible changes. We address this by using *scenario classification* to stratify the selection of scenarios.

Scenario Classification Strategies

The change scenarios cannot be selected randomly, thus the selected profile cannot represent the infinite number of all possible changes. But, we are not really interested in all possible changes, rather we want to have a representation of the set of likely changes which is a subset of all possible changes. The idea is to guide the selection to find at least one instance for each of the likely equivalence classes and thus make the scenario profile representative for all likely changes. In the case where each equivalence class get many scenarios, it will show as scenarios

that seem very similar. In such cases you either merge them into one, or select the one that is more likely to really occur.

In addition to the selection criterion, we also need a stopping criterion to decide when we have a representative set of scenarios and may stop collecting more. The stopping criterion derives from the change scenario classification scheme, such that: we continue collecting scenarios until a complete coverage of the classification scheme has been obtained.

In general, we can employ two approaches for selecting a set of scenarios: top-down and bottom-up.

Top-Down or Heuristic

The top-down approach uses a predefined classification of change categories to guide the search for change scenarios. This classification may be derived from analysis of the application domain, experience with potentially complex scenarios, heuristics or other sources. In interviews with stakeholders, the analyst uses this classification scheme to stimulate the interviewee to bring forward relevant scenarios. This approach is top-down, because we start with high-level classes of changes and then descend to concrete change scenarios.

Bottom-up or Case Specific

The bottom-up approach does not have a predefined classification scheme, and the stakeholders are first being interviewed without guidance of a predefined classification. The classification is then based on the change scenario collection. Then the collected scenarios are organized in these categories and the stakeholders review the categorized profile either to acknowledge the profile or add, modify or remove scenarios. This approach is bottom-up, because we start with concrete scenarios and then move to more abstract classes of scenarios, resulting in an explicitly defined and populated set of change categories.

In practice, we often combine the approaches, such that the change scenarios from interviews are used to build or refine our classification scheme. This (refined) scheme is then used to guide the search for additional scenarios. We have found a sufficient number of change scenarios when: (1) we have explicitly considered all identified change categories and (2) new change scenarios do not affect the classification structure.

'Good' Scenario Profiles

The elicitation technique to use depends on the characteristics of the set of change scenarios that is desired, i.e. the selection criterion. The selection criterion for scenarios is closely tied to the goal we pursue in the analysis:

- If the goal is to estimate maintenance effort, we want to select scenarios that correspond to changes that have a high probability of occurring during the operational life of the system.
- If the goal is to assess risks, we want to select scenarios that expose those risks.
- If the goal is to compare different architectures, we follow either of the above schemes and concentrate on scenarios that highlight differences between those architectures. If the candidates are very different and have different architecture styles, differences could be highlighted by scenarios that address known weaknesses in either of the candidates' styles. If the candidates are very similar, the scenarios that highlight differences could be scenarios describing extreme or rare situations.

Scenario Collection

Scenario profiles can be created in at least four organisational approaches. First, an individual could be assigned the task of independently creating a scenario profile for a software quality attribute of a system. Second, a group of people could be assigned the same task. Third, a group of people could be assigned the same task, but are required to prepare themselves individually before meeting with the group. Fourth, the group of people prepare only individual profiles that are merged without a meeting.

In the case of an individual creating a scenario profile, the advantage is, obviously, the relatively low resource cost for creating the profile. However, the disadvantage is that there is a risk that the scenario profile is less representative due to the individual's lack of experience in the domain, or misconceptions about the system.

The second alternative, i.e. a group that jointly prepares a scenario profile, has as an associated disadvantage that the cost of preparing the profile is multiplied by the number of

members of the group, meaning that the profile creation gets maybe three to five times more expensive. However, the risk of the forecast being influenced by individual differences is reduced since the group has to agree on a profile. Nevertheless, a risk with this method is that the resulting scenario profile is influenced by the most dominant rather than the most knowledgeable person, and thus affecting the scenario profile negatively. Finally, the productivity might be very low when in group session, since obtaining group consensus is a potentially tedious process.

The third alternative, in which the group members prepare an individual profile prior to the group meeting, has as an advantage that the individual productivity and creativity is incorporated when preparing the profiles, and then the unwanted variation of individuals are reduced by having the group agreeing on a merged scenario profile. A disadvantage is the increased cost, at least when compared to the individual case, but possibly also when compared to the unprepared group alternative.

The fourth alternative, in which individuals prepare scenario profiles that are then merged without a meeting, reduces the variance of using only individuals and lessen the cost by removing the need for a meeting.

As the reader understand there is a number of uncertainties as to which assumptions of advantages and disadvantages are correct. In the next chapter we describe the design and results of a controlled experiment on the above issues.

Conclusions

In this chapter we have discussed the elicitation process and its effects on the analysis results. In scenario-based analysis methods the scenarios and especially the scenario elicitation are extremely important for the results. The problem is similar to that of research methods, i.e. the validity of the results are dependent on the the process and control that led to the conclusions. Because we cannot verify that the change profile contains only and all scenarios that will occur, the confidence in the results stem from the control over the elicitation process.

To ensure interpretable results, elicitation require both a selection criterion, i.e. a criterion that states which scenarios are

important, and a categorization and stopping criterion, i.e. a criterion that determines when we have found sufficient scenarios.

The selection criterion is directly deduced from the goal of the analysis. To find scenarios that satisfy this selection criterion, we classify scenarios. We have distinguished two approaches to scenario elicitation: a top-down approach, in which we use a classification to guide the elicitation process, and a bottom-up approach, in which we use concrete scenarios to build up the classification structure. In both cases, the stopping criterion is inferred from the classification scheme.

Further, we have identified a number of organisational approaches for collecting the scenarios; an individual creating the scenario profile, unprepared groups create the scenario profile, individually prepared groups create the scenario and multiple individuals each create scenario profiles that are merged.

CHAPTER 5: **An Experiment on Scenario Elicitation**

Scenario-based software architecture assessments can only be as good as the scenario profiles used. If the wrong scenarios are analyzed, the results from the assessment does not apply to the intended architecture and period. Also, analyzing superfluous scenarios is a waste of valuable time. Hence, when managing scenario-based software architecture assessments it is crucial to obtain relevant and valid scenarios with the resources at hand. Typically, questions like the following needs answers based on objective empirical studies:

- Should we use a group?
- What size should the group be?
- Is larger always better?
- Is it necessary to have the meeting or is it equally well to just synthesize the individual profiles?
- What would happen if our local hero is present?
- What if we put together an all-star group?

These questions all address the issues of improving the process of scenario elicitation and avoiding unnecessary spending of valuable persons' time.

One can identify three categories of architecture assessment techniques, i.e. scenario-based, simulation and static model-based assessment. However, these techniques all make use of scenario profiles, i.e. a set of scenarios. Simulation and statical model analysis use scenarios to define the parameter values to use with the model. Scenario-based analysis of modifiability, for example requires a change profile containing a set of change scenarios.

The intention of the experiment is threefold:

- 1 testing four different methods of synthesizing scenario profiles.
- 2 test the hypothesis that there is a difference between the methods.
- 3 find out which one of the four methods that is better.

In our work with the design of this experiment we used the framework for experimentation in software engineering presented in (Basili *et al.* 1986). To conduct the experiment, we used volunteering students from the Software Engineering study program at the Blekinge Institute of Technology. The students participated in the experiment as an extra-curricular activity, and varied from 3rd-5th year students. The experiment was not conducted as part of a class or course.

After the experiment was designed and conducted, we discovered the opportunity to extend the planned analysis of the data by using virtual groups. The description of the technique, the virtual groups analysis and the conclusions are added to this chapter.

Experiment Design

Goal and purpose

The purpose of this experiment is to gain understanding of the characteristics of scenario profiles and the influence and sensitivity of individuals participating in the specification of the scenario profiles. The questions we are asking and would like to answer are:

- How much do profiles created by independent persons vary for a particular system?
- How does a profile, created by an independent person, differentiate from a profile created by a group?
- What are the difference in the results from scenario profile created by a group, if the individual members have prepared their own profiles first, compared to profiles created groups with unprepared members.

- How do these variances impact the predicted values? Are they absolutely critical to the method?

In the next section these questions have been formulated as more specific hypotheses and corresponding null-hypotheses.

Hypotheses

We state the following null-hypotheses:

H_{01} = *No significant difference in score between scenario profiles created by individual persons, and groups with unprepared members.*

H_{02} = *No significant difference in score between scenario profiles created by individual persons, and groups with prepared members.*

H_{03} = *No significant difference in score between scenario profiles created by groups with unprepared members, and groups with prepared members.*

In addition we state our six main hypotheses that allow us to rank the methods, even partially if the experiment does not produce significant results to support all stated hypotheses:

H_1 = *Scenario profiles created by groups with unprepared members generally get better scores than scenario profiles created by an individual person.*

And the counter hypothesis to H_1 , denoted H_{10} to more clearly show its relation to H_1 .

H_{10} = *Scenario profiles created by individuals generally get better score than profiles created by groups with unprepared members.*

H_2 = *Scenario profiles created by groups with prepared members, generally get better scores than scenario profiles created by an individual person.*

H_{20} = Scenario profiles created by individuals generally get better score than profiles created by groups with prepared members.

H_3 = Scenario profiles created by groups with prepared members generally get better scores than group profiles with unprepared members.

H_{30} = Scenario profiles created by groups with unprepared individuals generally get better score than profiles created by groups with prepared members.

These hypotheses will allow us to make some conclusions about the ranking between the methods, even though the data does not allow us to dismiss all null-hypotheses or support all the main hypotheses.

Treatments

To test these hypotheses using an experiment, we decided to employ a blocked project design with two project requirement specifications and twelve persons divided into four groups with three persons in each group.

The three ‘treatments’ that we use in the experiment are the following:

- 1 One independent person create a change scenario profile.
- 2 A group, with unprepared members, create a change scenario profiles.
- 3 A group, with members prepared by creating personal profiles before meeting in the group, creates the change scenario profile.

One of the problems in executing software development experiments is the number of persons required to test different treatments in robust experiment designs. In our previous experimentation experience, our main problem has been to find sufficient numbers of voluntary participants. This is optimized in the design of the experiment by having the group members prepare their own scenario profile that is collected and distributed before the group meeting is held and the group creates the group scenario profile. Thus, data for treatment 1 is

collected as part of treatment 3. This way we reduce the required number of subjects by half.

Analysis of Results

In order to confirm or reject any of the hypotheses, we need to rank the scenario profiles. The ranking between two scenario profiles must, at least, allow for deciding whether one scenario profile is better, equivalent, or worse than another scenario profile. The problem is that the profile is supposed to represent the future maintenance of the system and hence, the best profile is the one that is the best approximation of that. In the case where historical maintenance data is available, we can easily rank the scenario profiles by comparing each profile with the actual maintenance activities. However, for the project requirement specifications used in the experiment, no such data is available.

Instead we assume that the consensus of all scenario profiles generated during the experiment, i.e. a synthetic reference profile, can be assumed to be reasonably close to a scenario profile based on historical data. Consequently, the reference profile can be used for ranking the scenario profiles.

Reference Profile

When conducting the experiment we obtain 20 scenario profiles divided on two projects, 12 individually created, and 8 created by groups. These scenario profiles share some scenarios and contain some scenarios that are unique. If we construct a reference profile containing all unique scenarios using the scenario profiles generated during the experiment, we are able to collect the frequency for each unique scenario. Each scenario in the reference profile would have a frequency between 1 and 10. Using the reference profile, we are able to calculate a score for each scenario profile generated by the experiment by summarizing the frequency of each scenario in the scenario profile. This is based on the assumption that the importance of a scenario is indicated by the number of persons who believed it to be relevant. Consequently, the most important scenario has the highest frequency. The most relevant scenario profile must be composed by the most relevant scenarios and thus render the highest score.

To formalize the above, we define the set of all scenario profiles generated by the experiment $Q = \{P_1, \dots, P_{20}\}$, where $P_i = \{s_1, \dots, s_n\}$. The reference profile R is defined as $R = \{u_1, \dots, u_m\}$ where u_i is a unique scenario existing in one or more scenario profiles P . The function $f(u_i)$ returns the number of occurrences of the unique scenario in Q , whereas the function $m(s_i)$ maps a scenario from a scenario profile to a unique scenario in the reference profile. The score of a scenario profile can then be defined as follows (equation 5):

Equation 5.

$$\text{score}(P_i) = \sum_{S_x \in P_i} f(m(s_x))$$

Virtual Groups

Virtual groups is a technique to extend the analysis possibilities for a set of individual data points by combining them into *virtual groups*. Virtual groups have been used in other empirical research in software engineering. Briand *et al.*(2000) used the technique to create virtual inspections when evaluating capture-recapture-based inspection fault estimators. Data from two inspection experiments were used to evaluate multiple capture-recapture models. The comparison was based on the relative error which was calculated from the number of faults found and the actual number of faults obtained by seeding the inspected documents.

Form virtual groups

The basic requirement is that the combination of the individual data points is meaningful. In our case, the combination is the union of the individual profiles, which, in essence, is the same as the prepared groups in the experiment without the meeting.

If one has n individual data points you may form groups of size 2 to n members. The number of k member groups that can be formed from n individual data points is given by the formula:

Equation 6.

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Which means that with six data points we can form twenty unique groups of three or fifteen unique groups of two. In our study we created virtual groups systematically from two to six members.

- Consequences** A drawback with the technique is that each individual data point appears in multiple groups of the same size. Because of this the results of the virtual groups are not completely independent. However, the problem is manageable since for all groups at all times we know which data points have been used to form each virtual groups.
- Strengths** A strength of the method is that it allows us to analyze group configurations that would require substantial amounts of effort and several more persons to obtain the same amount of data. Of course, such data would allow us to do slightly more analysis, but virtual groups allow is to gain insight that can be used to draw conclusions, direct and optimize future studies.

The Selected Projects

Two requirements specifications have been selected from two different projects. Project Alpha is the requirements specification of the prototype for a successor system of a library system called BTJ 2000. This system is widely used in public libraries in Sweden. The system is becoming old-fashioned and needs to be renewed to enter the market of university libraries.

The requirements specification of project Beta defines a support and service application for haemo dialysis machines. The new application shall aid service technicians in error tracing of erroneous system behavior and in doing diagnostics on the system for fault prevention.

Both projects have been developed for commercial companies as customers and represent commercial software applications. In fact, one of the projects resulted in a ready product that has been included in the product portfolio of the customer.

Operation

The experiment is executed according to the following steps: (schedule in table 3)

- 1 A set of individuals with varying programming and design experience are selected.
- 2 A presentation of the method is given as part of the experiment briefing. A document describing the method is also available for all to study during the experiment.
- 3 Each person fills in an experience form.

- 4 Individuals are assigned to groups of three using the matched pairs principle based on the experience as captured by the personal information form (see page 91).
- 5 The groups are assigned a 'treatment' and the requirement specification for the first project is handed out. The group A and B that are assigned to the prepared group profile method, start on individual basis which is part of both treatment 1 and treatment 3.
- 6 When 1.5 hours of time have passed the profiles of the individuals are collected during a short break. During the break the individual profiles are photocopied and handed back to the respective authors. Great care must be taken in that the profile returns to the correct person without any other person getting a glimpse.
- 7 After the break groups A and B continue in plenum and each group prepares a group scenario profile.
- 8 At noon, all the group profiles are collected and groups proceed to lunch.
- 9 After lunch, the process is repeated from step 5, but groups A and B now produce a group profile from start, and groups C and D begin with preparing an individual scenario profile before proceeding in plenum to produce their respective group scenario profile.

Table 3: Experiment One Day Schedule

Time	Group A/B	Group C/D	Project
08.00	Introduction and experiment instructions		
09.00	Individual Profile Preparation	Unprepared Group	Alpha
10.30	Prepared Group		
12.00	LUNCH BREAK		
13.00	Unprepared Group	Individual Profile Preparation	Beta
14.30		Prepared Group	
16.00	De-briefing		

All information collected during the experiment is tracked by an identification code also present on the personal information form. Consequently, the data is not anonymous, but this is, in our judgement, not an immense problem since the collected data is not, in any clear way, directly related to individual performance. Instead being able to identify persons that had part in interesting data points is more important than the risk of getting tampered data because of lack of anonymity.

Data Collection

The data collection in this experiment is primarily to collect the results of the work performed by the participants, i.e. the scenario profiles. However, some additional data is required, for example, a form to probe the experience level of the participants. The following forms are used:

- personal information form (Appendix A)
- individual scenario-profile form (Appendix B)
- group scenario-profile form (Appendix C)

The forms have been designed and reviewed with respect to gathering the correct data and ease of understanding, since misunderstanding the forms pose threats on the validity of the data. The personal information form is filled in by the participants after the introduction and collected immediately. The others forms are collected during the experiment. During the experiment briefing on all forms is done.

The Personal Scenario Profile form (Appendix B) is handed out to the experiment subjects at the beginning of the Individual Profile Preparation activity. It will be collected at the end of the activity and photocopies will be made for archives.

The Group Scenario Profile form (Appendix C) is handed out to the groups, along with the respective individuals completed profile form, at the start of the Group Synthesis Consensus activity.

Threats

External Threats

Some external threats can be identified in the experiment design, e.g. differences in experience and learning effects. For

the most part of the identified threats measures have been taken to eliminate these by adapting the design. By using the blocked project design we eliminate, for example, the risk of learning effects, and in the case of differences in participants experience we use the matched pairs technique when composing the groups, to ensure that all groups have a similar experience profile.

Although precautions have been taken in selecting a system from a large student project which is the result of a 'real' customers demands, this cannot absolutely exclude that the system is irrelevant. The industry customers often use this kind of student projects as proof of concept implementations. However, there are no reasons for the scenario profile prediction method not to be applicable in this situation like the above mentioned. And for the purpose of the experiment we feel that it is more crucial to the results that the individuals in the project have no experience with the particular system's successors.

Internal Threats

The internal validity of the experiment is very much dependent on the way we analyze and interpret the data. During the design, preparation, execution and analysis of the experiment and the experiment results, we have found some internal threats or arguments for possible internal validity problems. We discuss them and their impact in the following subsections.

Ranking Scheme When choosing the ranking scheme that we did for the analysis of the experiment we made an *important assumption*, i.e. the number of profiles containing a particular scenario is an acceptable indicator of its relevance. However, if this assumption cannot be accepted two threats to the ranking scheme may exist. First, the reference profile will be relative to the profiles since it is based on them. Second, there might be one single brilliant person that has realized a unique scenario that is really important, but since only one profile included it, its impact will be strongly reduced.

The first problem appears in the case where there are significant differences between the individually created profiles and the group profiles, the differences will be normalized in the reference profile. Given that the individually prepared profiles are more diverse than the profiles prepared by groups, those

profiles will render on average lower scores, while the group profiles will render on average higher scores. In case the results of the experiment is in favor to the null-hypothesis, we will not be able to make any distinction between the group prepared profiles or the individually prepared profiles ranking scores.

The second problem can be dealt with in two ways. First we can make use of the delphi method or the wide band delphi (Boehm 1981). In that case we would simply synthesize the reference profile, distribute it and have another go at the profiles and get more refined versions of the profile. The second approach is to make use of the weightings of each scenario and make the assumption that the relevance of a scenario is not only indicated by the number of profiles that include it, but also the weight it is assigned in these profiles. The implication of this is that a scenario that is included in all 20 of the profiles but has a low average weighting, is relevant but not significant for the outcome. However, a scenario included in only one or a few profiles is deemed less relevant by the general opinion. If it has a high average weighting, those few consider it very important for the outcome. Now, we can incorporate the average weighting in the ranking method by defining the ranking score for a profile as the sum of rank products (the frequency times the average weighting) of its scenarios. This would decrease the impact of a commonly occurring scenario with little impact and strengthen the less frequent scenarios with higher impact.

In this thesis, we have not addressed this threat since it is rid by the assumption we have stated clearly and explicitly to the reader.

**Technique itself
based on
hypothesis**

The ranking of profiles is based on the assumption that frequent scenarios are more important and, thus, lead to higher scores. One could suspect that the ranking technique is biased towards prepared groups and consequently implicitly favors the hypotheses we hope to confirm.

The way this bias would impact is that the participants prepare their individual scenario profiles and then the group only take the super set of the three individual scenario profiles. However, this does not just benefit the score for the prepared group profile, but also the individual profiles of the group members. Since both profile types benefit the same it does not influence the outcome of the experiment.

**Bias for quantity
instead of
quality.**

It could be the case that a profile reaches a high score by using many, unimportant scenarios. Some scenarios may not even be related to the project. This profile, that intuitively should obtain a low score, scores higher than a profile with fewer, but more important scenarios.

When we examine the example closer, we find that the first profile in the example could render a maximum score of 60, because of the limitation on six categories and ten scenarios in each category. A profile with only ten scenarios would have to score on average more than six per profile to out rank the long profile. In the first profile we would get a ratio of the number of scenarios in the profile and the score for that profile of exactly one. In the other profile example, the ratio would be more then one. In table 9 and 10, the ratios are presented for the projects. Thus we can detect if this indeed becomes a problem.

Concluding, although this may, theoretically, be an internal validity threat, it did not occur in the experiment reported in this paper.

**Reference Profile
Coding bias**

To create the reference profile all scenarios are put together in a table, i.e. the union of all profiles. Since scenarios may be equivalent in semantics in spite of being lexically different, the experimenter needs to establish what scenarios are equivalent, i.e. coding the data. The coding is done by taking the list of scenarios and for every scenario check if there was a previous scenario describing a semantically equivalent situation. This is done using the database table and we establish a reference profile using a frequency for each unique scenario.

The possible threat is that the reference profile reflects the knowledge of the person coding the scenarios of all the profiles, instead of the consensus among the different profiles. To reduce the impact of this threat, the coded list has been inspected by an additional person. Any deviating interpretations have been discussed and the coding have been updated according to the consensus after that discussion.

Analysis & Interpretation

In the previous section, the design of the experiment was discussed. In this section, we report on the results of conducting the experiment. We analyze the data by preparing the reference profiles, calculating the scores for all profiles and

determining average and standard deviations for each type of treatment. Further, we form and analyze the virtual groups. Finally, the stated hypotheses stated are evaluated.

Mortality

The design of the experiment requires the participation of 12 persons for a full day. For the experiment we had managed to gather in excess of 12 voluntary students, with the promise of a free lunch during the experiment and a nice *à la carte*-dinner after participating in the experiment. Unfortunately, some volunteering students did not show up at the time of the experiment. As a result, the experiment participants were only nine persons. Instead of aborting the experiment, we chose to keep the groups of three and to proceed with only three groups, instead of the planned four. As a consequence, the data from the experiment is not as complete as intended (see table 6 and 7). But nevertheless, we feel that the collected data is useful and allow us to validate our hypotheses and make some interesting observations.

Once the experiment had started, we had no mortality problems, i.e. all the participants completed their tasks and we collected the data according to plan.

Reference profiles

During the experiment we collected 142 scenarios from 9 profiles for project alpha, 85 scenarios from 6 profiles for project beta, totalling 227 scenarios from 15 profiles. The scenarios were coded with references to the first occurring equivalent scenario using a relational database to later generate one reference profile per project. The reference profile for project alpha included 72 scenarios and project beta included 39. The top 10 and top 8 scenarios of the reference profiles are presented in table 4 and 5. In the alpha case, we note that one scenario has been included in all nine profiles, i.e. has the score nine. In the beta project, we note that the top scenario is included seven times in six profiles. This could be an anomaly, but when investigated, we recognized that in one of the profiles from the beta project two scenarios have been coded as equivalent. This is probably not the intention by the profile creator, an individual person in this case, but we argue that the two scenarios are only slightly different and should correctly be coded as equivalent to the same scenario in another profile.

Table 4: Alpha Reference Profile Top 10.

Description	Frequency
new DBMS	9
new operating system on server	7
new version of TOR	7
introduction of smart card hardware	5
additional search capabilities	5
pureWeb (cgi) clients	4
support for serials	4
new communication protocol	4
user interface overhaul	4
new java technology	4

Table 5: Beta Reference Profile Top 8.

Description	Frequency
remote administration	7
upgrade of database	6
upgrade of OS	6
real-time presentation of values	4
change of System 1000 physical components (3-4 pcs.)	4
rule-based problem-learning system	3
change from metric system to american standard	3
new user levels	3

Another interesting observation to make is that among the top three scenarios in both projects we find changes of the database management system and changes of the operating systems, either new version or upgrade. We find this same scenario in just about all the profiles. This suggests that these two changes to a system are among the first scenarios that come to mind when thinking about future changes to a system. It does not, however, allow us to conclude that these changes are indeed the most commonly occurring ones.

Table 6: Project Alpha.

Identity	Members	Profile Score	Remarks
group A	C,D,E	72	individual preparation
group B	H,I,F	77	individual preparation
group C	A,B,G	49	no individual preparation
Arthur			only participated in a group
Bertram			only participated in a group
Charlie		39	
David		38	
Ernie		45	
Frank		19	
Gordon			only participated in a group
Harald		66	
Ivan		55	

Finally, it is worth noting that the major part of the top scenarios are related to interfacing systems or hardware. Only a few of the scenarios in the top 10 or 8 are related to functionality specific to the application domain, e.g. “support for serials” in table 4 or “new user levels” in table 5.

Ranking and Scores

In this section, we present the coded and summarized data collected from the experiment. In the table presented in table 6 the score for each of the profiles generated for the Alpha project are presented. It is interesting that group A and B, that both are prepared groups, score strikingly high scores, compared to the other profiles in project Alpha. Further, we notice little difference between the profiles created by the individual persons and the unprepared groups, in neither project.

In table 7, the profile scores for project Beta are presented. The prepared group, C in this case, scores very high, but the unprepared groups, A and B, score less. This is interesting since the groups members are the same for both projects.

Table 7: Project Beta

Identity	Members	Profile Score	Remarks
group A	C,D,E	44	no individual preparation
group B	H,I,F	37	no individual preparation
group C	A,B,G	72	individual preparation
Arthur		36	
Bertram		43	
Charlie			only participated in a group
David			only participated in a group
Ernie			only participated in a group
Frank			only participated in a group
Gordon		33	
Harald			only participated in a group
Ivan			only participated in a group

In table 8 the average score for each type of treatment is presented for the Alpha, Beta project and in total. In addition, the standard deviation over the scores and the number of cases is presented. The average score for prepared groups is substantially higher than the score for unprepared groups or individuals. Secondly, the standard deviation is the largest for individuals, i.e. 13, but only 6 for unprepared groups and 3 for prepared groups. Finally, it is interesting to note that the standard deviation for all profiles is larger than for any of the treatments, which indicates that the profiles for each type of treatment are more related to each other than to profiles for other treatment types.

Previously, we discussed various threats to the internal validity of the experiment. One of the discussed threats is the risk that a profile with many unimportant scenarios scores higher than a profile with fewer, but more important scenarios, while this is

Table 8: Average and Standard Deviation Data

Treatment	Alpha	Beta	Total	Std. Dev.	#cases
Individual	43	37	41	13	9
Unprepared group	49	40	43	6	3
Prepared group	75	72	74	3	3
Total	51	44	48	17	15

counter intuitive. Based on the data in table 9 and 10, we can conclude that although a theoretical threat was present, it did not occur in the experiment.

Table 9: Project Alpha

Identity	Profile Score	Identity	Average Score	Identity	Profile Length
group B	77	Ernie	4,5	group B	23
group A	72	Harald	3,9	group A	19
Harald	66	group A	3,8	Ivan	19
Ivan	55	Charlie	3,5	group C	18
group C	49	group B	3,3	Harald	17
Ernie	45	David	3,2	Frank	13
Charlie	39	Ivan	2,9	David	12
David	38	group C	2,7	Charlie	11
Frank	19	Frank	1,5	Ernie	10

Table 10: Project Beta

Identity	Profile Score	Identity	Average Score	Identity	Profile Length
group C	72	Arthur	3,6	group C	24
group A	44	group A	3,4	group A	13
Bertram	43	Bertram	3,3	group B	13
group B	37	group C	3,0	Bertram	13
Arthur	36	Harald	2,8	Harald	12
Harald	33	group B	2,8	Arthur	10

Evaluating the Hypotheses

The experiment data does not allow us to identify any significant difference in ranking between profiles created by an independent person or profiles created by a group with unprepared members. Hence we *cannot dismiss* the null hypothesis, H_{01} .

The first null hypothesis, H_{01} , counters the two hypotheses H_1 and H_{10} . Since the experiment data does not allow us to dismiss the null hypothesis H_{01} , we cannot expect to validate those two hypotheses and therefore, we can *dismiss* H_1 and H_{10} . We can, however, make an *interesting observation* on the variation in the ranking scores between the profiles of the individuals and the unprepared groups. The scores of the profiles created by independent persons range from 19 to 62 over both projects, while the scores of the profiles created by the unprepared groups only ranges from 32 - 49 over both projects. The observation is also supported by the standard deviation values presented in table 8. This suggests that using unprepared groups does not lead to higher scores on the average, but provides more stable profiles and reduces the risk for extreme results, i.e. outliers.

With respect to the second null hypothesis, H_{02} , we find strong support in the analyzed data for a significant difference between the profiles created by individuals, with an score average of 43, and profiles created by a group with prepared members, with an score average of 74 (see table 8). We also observe that no profile created by an independent person has scored a higher score than any profile created by a group with prepared members. Hence, we can *dismiss* the second null hypothesis, H_{02} .

Because we were able to dismiss the second null hypothesis, it is worthwhile to examine the two related hypotheses, H_2 and H_{20} . The scores clearly show that the group with prepared members in all cases have scored higher than the profiles created by independent persons. This allows us to *confirm* the hypothesis, H_2 and allow us to *dismiss* the counter hypothesis, H_{20} .

The last null-hypothesis is H_{03} . With respect to this hypothesis, we find support that a significant difference exists between the average scores for unprepared and prepared groups. Profiles created by groups with prepared members score 74 on average,

as opposed to profiles from groups with unprepared members, that score 41 on average. Hence, we *can dismiss* the null hypothesis, H_{03} , and evaluate the related hypotheses H_3 and H_{30} . The average score for prepared groups is 74, which is considerably higher than the average score for unprepared groups, i.e. 41. Based on this, we are able to *confirm* hypothesis H_3 and, consequently, *dismiss* the counter hypothesis H_{30} .

Analysis Based on Virtual Groups

In this section we first discuss the group sizes, then the synergy effects of the meetings, and finally the influence of certain individuals in the groups. Because of this we focus our analysis on project alpha and turn to the data in project beta for getting support on conclusions.

Truncating the Reference Profile

Also, we are interested in determining if the group size affect the ratio between scenarios that are relevant, i.e. more than one person listed it, and scenarios that are considered irrelevant, i.e. only one person listed the scenario. To focus the results towards this end we remove, from the reference profile, all scenarios with the score one. Thus, we get two groups of scenarios; one group of scoring scenarios and one group of non-scoring scenarios. This will allow us to study differences between profiles in terms of wasting time on analyzing irrelevant scenarios.

A concern is that there could be a small chance of one person coming up with a scenario that is very relevant and no one else does. However, the experiment subjects had very similar level of domain expertise and based on this the chance that one person would come up with a scenario that is truly brilliant and no one else would is estimated as extremely small, and therefore negligible.

Results of the Virtual Groups

Appendix D holds the profile data from the respective projects (Table 1 and 2). The original individual profiles are marked with an asterisk(*). The results are sorted on number of members, then on the profile score and then on the number of scenarios in the profile that was also in the reference profile (# in Ref.), i.e. scoring scenarios. In this way when there is a tie

between two or more profiles we list the one with highest score per scenario first.

Deciding the group size

In figure 14 and 15 we present a scatter plot of the pairs {Profile Score, # of Members} for the individuals, physical groups and the virtual groups. The score of the virtual groups hits the ‘roof’ already at four members. The reason is that all these groups cover all scenarios in the reference profile.

Figure 14.

Score versus # of members - Project Alpha

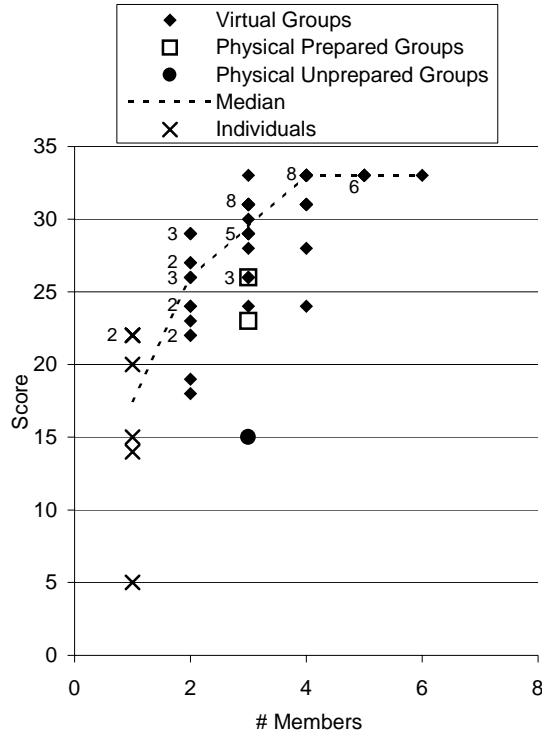
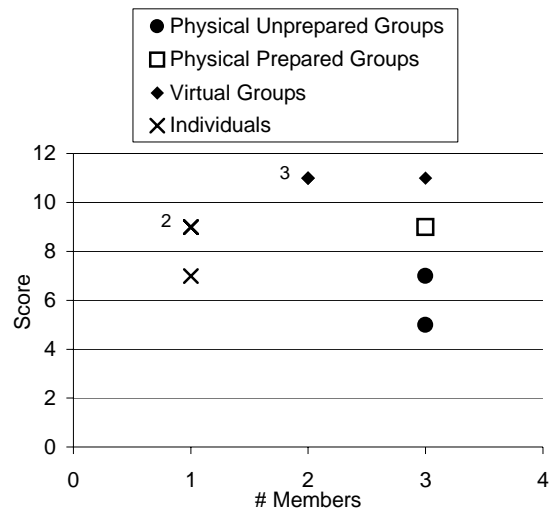


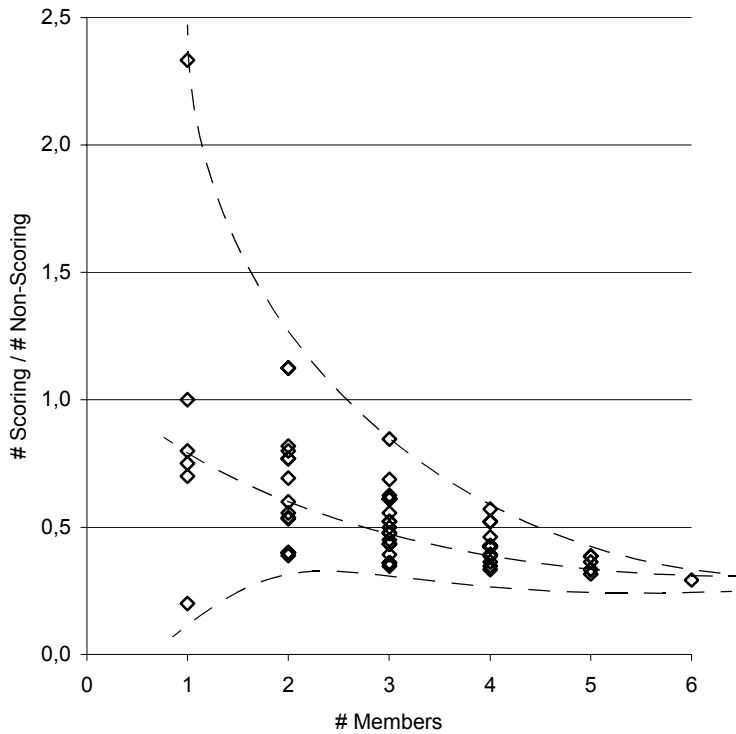
Figure 15.
*Score versus # of
members - Project
Beta*



However, the efficiency of the profiles might not be the same between the highest scoring profiles, i.e. two profiles that score max may have different lengths. Analyzing scenarios costs money and time, hence, the profile to prefer is the one with the highest score and lower number of scenarios. In figure 16 the diagram shows the ration between the numbers of scoring and non-scoring scenarios in each profile for each group size. There is one extreme value for the individuals that only have 2,5 scoring scenarios for each non-scoring scenario. It also happens to be the best scoring individual profile. What we can observe from the diagram is that the variance is greater for smaller groups and individuals. But, the decreased variance inevitably means decreasing rations, such that for every scoring scenario we get two more non-scoring scenarios. This clearly indicates that the problem when increasing group size, is the need to deal with scenarios that are irrelevant to all members but one. The point being, that for one additional relevant scenario you receive *more* than one irrelevant scenario. And clearly, in this case, going from four to five person groups is just waste of time.

Figure 16.

The profile ratio between scoring and non-scoring scenarios versus the group size.



Synergy effects of the meeting

The meeting's effect on the profile score of the prepared groups were, to our surprise, negative in all cases (Table 11). We had expected that the *real* prepared groups scored the same as the virtual unions did, but this was not the case.

Pruned Profiles

Apparently the groups have pruned the scenario profile during the meeting and removed valuable scenarios from their joint profile. In project alpha, the groups removed 7 and 17 scenarios and of those 2 and 4 respectively, are present in the reference profile. This was probably because the scenarios were championed only by one member of the real group, but persons outside the group also had them in their profile. This means that the pruning is effective but has in this case five to twenty percent risk of rejecting a valuable scenario. The meeting had a positive effect on the efficiency of the profile, since most of the scenarios that were removed improved the relation between scoring and non-scoring scenarios. The first group improved the profile to more than one scoring scenario per non-scoring scenario and the other group brought the relation closer to 1 scoring scenario per two non-scoring.

Table 11: Physical groups versus virtual groups - project alpha

	Group 1 Profile		Group 2 Profile	
	Physical	Virtual	Physical	Virtual
Id	1	300	2	319
Score	26	30	23	31
Score Diff.	4		8	
Length	17	24	22	39
# Scoring / # Non-Scoring	1,13	0,85	0,47	0,39
Diff.	0,28		0,08	

Added Scenarios During the two meetings only two and three new scenarios were added to the groups profile, i.e. scenarios that none of the members had in their profile before the meeting. Hence we can conclude that the synergy effect of the persons being inspired by the other persons profiles to come up with additional scenario that one intuitively would expect only occurs on a very limited basis.

Individuals

In project alpha one person, David, is in all the top profiles. But David is not the best scoring individual profile. In fact David’s personal profile only scored the fourth highest.

The highest scoring individual profile in project alpha is only part of one of the top scoring profiles, the four member top profile. The reason could be that to score the highest with an individual profile, you need to include the scenarios that others do, too. In doing so you automatically make your profile less valuable to the group, since in essence the scenarios you bring to the groups have already been thought of. For the group it is more valuable if you bring a set of scenarios that were not thought of by the fellow members but present in the reference profile.

However, in table 12 we can compare the median score for all the groups of a certain size to the median score of the groups where Ernie was a member and the same for Frank. Ernie scored highest with his individual profile, and the groups where he has been a member consistently have higher or same median score

than all the groups of the same size. In fact, the opposite is also true. Frank made a really bad profile, and the groups where he was a member scored lower median scores than the median for all the groups of the size. From this we can conclude that a member can contribute significantly, but, the influence is reduced with increased group size.

Table 12: Influence of best and worst profile

# Members	Median Score		
	All	With Ernie	With Frank
1	17,5	22	5
2	26	27	22
3	29,5	31	29
4	33	33	31
5	33	33	33

Conclusions

In this chapter we have presented the design and results of an experiment on four methods for creating scenario profiles. The methods, or treatments, for creating scenario profiles that were examined are (1) an individual prepares a profile, (2) a group with unprepared members prepares a profile, (3) a group with members that, in advance, created their individual profiles as preparation.

We also have stated a number of hypotheses, with the corresponding null-hypotheses and, although, the results of the experiment data do not allow us to dismiss all of our null-hypotheses, we find support for the following hypotheses:

H_2 = Scenario profiles created by groups with prepared members generally get better scores than scenario profiles created by an individual person.

H_3 = Scenario profiles created by groups with prepared members generally get better scores than group profiles with unprepared members.

Thus, based on the first analysis of the experiment data, we are able to conclude that using groups with prepared members is the preferred method for preparing scenario profiles.

The analysis based on virtual groups provided the following findings:

- Prepared group meetings may easily affect the joint scenario profile score negatively compared to the virtual groups' profiles and add few additional scenarios. Instead the group uses the meeting to prune the profile, i.e. removing irrelevant scenarios, and decreases the number of irrelevant scenarios in the scenario profile. This supports the conclusion that prepared groups are to prefer, because the efficiency of the profile is likely to be higher.
- The size of the group in this case has an optimum between the added relevant scenarios you get and the added irrelevant scenarios. This optimum may vary from case to case, but in this case it was between two and three member groups. Clearly, choosing larger groups than three, in this case, is very hard to motivate.
- Individual performance can be shown to affect the group performance. The effect is limited though, and it is not required for a group that the best scoring individual is a member of the group in order to achieve a high score. Hence, we can conclude that it is more important to have a group of people, than making sure that the local hero is present in the group.

In addition we have made the following observations during the experiment and during the analysis of the data:

- 1 Two change scenarios occurring in just about all the profiles were new version or upgrade of the database management system and the operating system.
- 2 Few scenarios among the top 10 or 8 are related to the application. Instead most of the key scenarios are related to interfacing systems or hardware.
- 3 The standard deviation in score is lower for profiles created by unprepared groups, than for individuals, although the average of the profiles scores cannot not be said to differ significantly between the two.

- 4 The unprepared groups score less than the median of the individual profiles and clearly below the prepared groups. With the synergy effect of the meeting as a means to increase the profile score being rejected, the synergy effects are likely to reside in the combination of the results of multiple minds rather than the combination of the minds themselves.

Based on this study we have identified the following issues that deserves more attention in future research on scenario-based assessment techniques:

- The study could benefit from having more individual profiles than six and three respectively. This would allow better possibilities to study what group sizes are better.
- The meetings have been studied here as a closed activity and no internal data from the meetings has been collected. To find out what really happens during the meetings in-depth studies of such meetings are highly motivated.
- The reference profile builds on the importance of consensus. A strength is that it is possible to use for other kinds of scenarios than change scenarios. On the other hand, what we would really like is to verify created profiles against the future it is said to predict. This verification problem is inherent to all research on scenario-based assessment, since at the time of the assessment the only success criteria available is that the participants and managers were happy with the results, or the process.

After performing the experiment and this extended study we are convinced that the scenario elicitation's influence on the assessment process cannot be under-estimated and that the need for in depth research to validate scenario-based assessment approaches are urgent.

PART 2:

Case Studies

Part II begins with a case study in chapter 6 describing many of the founding experiences for the work presented in this thesis. The following case studies are all applications of the modifiability analysis model as presented in Part I. Chapter 7 is the earliest application of the method and was mainly a result from the case study in chapter 6. Chapters 8, 9 and 10 describe later applications of a more recent version of the analysis method. Chapter 11 concludes Part II and presents the overall experiences from the case studies concerning applying the modifiability analysis method.

CHAPTER 6: **Haemo-Dialysis Case**

This chapter presents the experiences and software architecture from a cooperative research project between Blekinge Institute of Technology, Althin Medical AB and EC-Gruppen AB. The research project's main goal was to design a new software architecture for the dialysis machines produced by Althin Medical AB. The current product generation's software was exceedingly hard to maintain and certify. EC-Gruppen's own goal in the project was to study novel ways of constructing embedded systems, whereas our goal was to study the process of designing software architecture and to collect experiences.

The next sub section presents the application domain, the legacy architecture and the typical requirements in the domain. The subsequent section presents the lessons we learned during the case study, followed by sections describing the software architecture and the evaluation. The last section in this chapter presents the conclusions.

Haemo Dialysis Machines

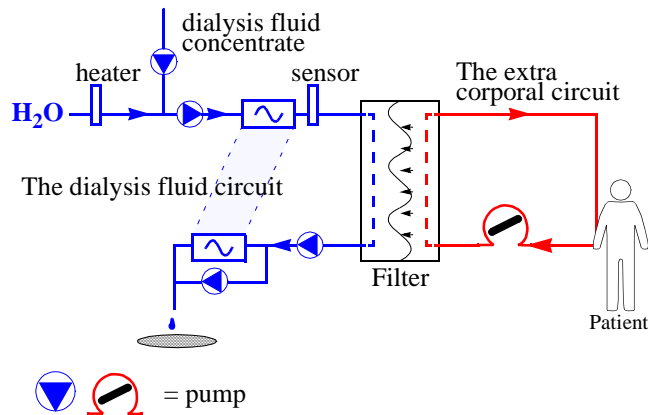
Haemo dialysis systems present an area in the domain of medical equipment where competition has been increasing drastically during recent years. The aim of dialysis systems is to remove water and certain natural waste products from the patient's blood. Patients that have, generally serious, kidney problems use this type of system to replace the natural process performed by the kidneys with an artificial one.

An overview of a dialysis system is presented in figure 17. The system is physically separated into two parts by the dialysis membrane. On the left side the dialysis fluid circuit takes the water from a supply of a certain purity (not necessarily sterile), dialysis concentrate is added using pumps. A sensor monitors the concentration of the dialysis fluid and the measured value is used to control the pump. A second pump maintains the flow

of dialysis fluid, whereas a third pump increases the flow and thus reduces the pressure at the dialysis fluid side. This is needed to pull the waste products from the patient's blood through the membrane into the dialysis fluid. A constant flow of dialysis fluid is maintained by the hydro mechanic devices that ensure exact and steady flow on each side (rectangle with a curl).

Figure 17 presents a schematic overview of a haemo dialysis machine. The extra corporal circuit, i.e. the filter-side connected to the patient's blood stream, has a pump for maintaining a specified blood flow on its side of the membrane. The patient is connected to this part through two needles usually located in the arm that take blood to and from the patient. The extra corporal circuit uses a number of sensors, e.g. for identifying air bubbles, and actuators, e.g. a heparin pump to avoid clotting of the patients blood while it is outside the body. However, these details are omitted since they are not needed for the discussion in the paper.

Figure 17.
*Schematic of Haemo
Dialysis Machine*



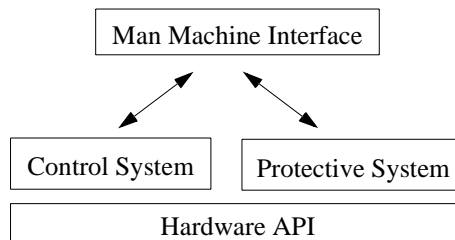
The dialysis process, or *treatment*, is by no means a standard process. There is a fair collection of treatments including, for example, Haemo Dialysis Filtration (HDF) and Ultra Filtration (UF) and other variations, such as single needle/single pump, double needle/single pump. Treatments are changed due to new research results but also since the effectiveness of a particular treatment decreases when it is used too long for the same patient. Although the abstract function of a dialysis system may seem constant, the set of variations impact the design and construction of the software. Our haemo dialysis producers anticipate that several changes to the software, hardware and

mechanical parts of the system will be necessary in the future in response to developments in medical research.

Legacy Architecture

As an input to the project, the original application architecture was used. This architecture had evolved from being only a couple of thousand lines of code very close to the hardware to close to a hundred thousands lines mostly on a higher level than the hardware API. The system runs on a PC-board equivalent using a real-time kernel/operating system. It has a graphical user interface and displays data using different kinds of widgets. It is a quite complex piece of software and because of its unintended evolution, the structure that was once present has deteriorated substantially. The three major software subsystems are the Man Machine Interface (MMI), the Control System, and the Protective system (see figure 18).

Figure 18.
*Legacy system
decomposition*



The *MMI* has the responsibilities of presenting data and alarms to the user, i.e. a nurse, and getting input, i.e., commands or treatment data, from the user and setting the protective and control system in the correct modes.

The *control system* is responsible for maintaining the values set by the user and adjusting the values according to the treatment selected for the time being. The control system is not a tight-loop process control system, only a few such loops exist, most of them low-level and implemented in hardware.

The *protective system* is responsible for detecting any hazard situation where the patient might be hurt. It is supposed to be as separate from the other parts of the system as possible and usually runs in its own task or process. When detecting a hazard, the protective system raises an alarm and engages a process of returning the system to a safe-state. Usually, the safe-state is stopping the blood flow or dialysis-fluid flow.

The documented structure of the system is no more fine-grained than this and to do any change impact analysis, extensive knowledge of the source code is required.

System Requirements

The aim during architectural design is to optimize the potential of the architecture (and the system built based on it) to fulfil the software quality requirements. For dialysis systems, the driving software quality requirements are *maintainability*, *reusability*, *safety*, *real-timeliness* and *demonstrability*. Below, these quality requirements are described in the context of dialysis systems.

Maintainability Past haemo dialysis machines produced by our partner company have proven to be hard to maintain. Each release of software with bug corrections and function extensions have made the software harder and harder to comprehend and maintain. One of the major requirements for the software architecture for the new dialysis system family is that maintainability should be considerably better than the existing systems, with respect to *corrective* but especially *adaptive* maintenance:

- 1 *Corrective maintenance* has been hard in the existing systems since dependencies between different parts of the software have been hard to identify and visualize.
- 2 *Adaptive maintenance* is initiated by a constant stream of new and changing requirements. Examples include new mechanical components as pumps, heaters and AD/DA converters, but also new treatments, control algorithms and safety regulations. All these new requirements need to be introduced in the system as easily as possible. Changes to the mechanics or hardware of the system almost always require changes to the software as well. In the existing system, all these extensions have deteriorated the structure, and consequently the maintainability, of the software and subsequent changes are harder to implement. Adaptive maintainability was perhaps the most important requirement on the system.

Reusability The software developed for the dialysis machine should be reusable. Already today there are different models of haemo dialysis machines and market requirements for customization will most probably require a larger number of haemo dialysis

models. Of course, the reuse level between different haemo dialysis machine models should be high.

Safety Haemo dialysis machines operate as an extension of the patients blood flow and numerous situations could appear that are harmful and possibly even lethal to the patient. Since the safety of the patient has very high priority, the system has extremely strict safety requirements. The haemo dialysis system may not expose the dialysis patient to any hazard, but should detect the rise of such conditions and return the dialysis machine and the patient to a state which present no danger to the patient, i.e. a safe-state. Actions, like stopping the dialysis fluid if concentrations are out of range and stopping the blood flow if air bubbles are detected in the extra corporal system, are such protective measures to achieve a safe state. This requirement have to some extent already been transformed into functional requirements by the safety requirements standard for haemo dialysis machines (CEI/IEC 601-2), but only as far as to define a number of hazard situations, corresponding threshold values and the method to use for achieving the safe-state. However, a number of other criteria affecting safety are not dealt with. For example, if the communication with a pump fails, the system should be able to determine the risk and deal with it as necessary, i.e. achieving safe state and notify the nurse that a service technician is required.

Real-timeliness The process of haemo dialysis is, by nature, not a very time critical process, in the sense that actions must be taken within a few milli- or microseconds during normal operation. During a typical treatment, once the flows, concentrations and temperatures are set, the process only requires monitoring. However, response time becomes important when a hazard or fault condition arises. In the case of a detected hazard, e.g. air is detected in the extra corporal unit, the haemo dialysis machine must react very quickly to immediately return the system to a safe state. Timings for these situation are presented in the safety standard for haemo dialysis machines (CEI/IEC 601-2).

Demonstrability As previously stated, the patient safety is very important. To ensure that haemo dialysis machines that are sold adhere to the regulations for safety, an independent certification institute must certify each construction. The certification process is repeated for every (major) new release of the software which substantially increases the cost for developing and maintaining the haemo dialysis machines. One way to reduce the cost for

certification is to make it easy to demonstrate that the software performs the required safety functions as required. This requirement we denote as *demonstrability*.

Lessons Learned

During the architecture design project, we gathered a number of experiences that, we believe, have validity in a more general context than the project itself. In this section, we present the lessons that we learned. In the next section, the foundations for and the process of architecture design leading to those lessons and experiences are presented.

Quality requirements without context

Different from functional requirements, quality requirements are often rather hard to specify. For instance, one of the driving quality requirements in this project was maintainability. The requirement from Althin Medical, however, was that maintainability should be “as good as possible” and “considerably better than the current system”. In other projects, we have seen formulations such as “high maintainability”. Even in the case where the ISO standard definitions (ISO 9126) are used for specifying the requirements, such formulations are useless from a design and evaluation perspective. For example, maintainability mean different things for different applications, i.e. haemo dialysis machine software and a word processor have totally different maintenance. The concrete semantics of a quality attribute, like maintainability, is dependent on its context. The functional requirements play an important role in providing this context, but are not enough for the designer to comprehend what actual maintenance tasks can be expected.

Based on our experience, we are convinced that quality requirements should be accompanied with some context that facilitates assessment. The nature of the context depends on the quality requirement. For instance, to assess maintainability, one may use a *maintenance profile*, i.e. a set of possible maintenance scenarios with an associated likelihood. To assess performance, one requires data on the underlying hardware and a *usage profile*. Based on such profiles, one is able to perform a relevant analysis of the quality attributes. Every quality requirement requires its own context, although some profiles, e.g., the usage profile, can be shared.

Since the customer had specified the quality requirements rather vaguely, we were forced to define these in more detail. We felt that the time needed to specify the profiles was well worth the effort. It serves as a mental tool for thinking about the real effects on the system and its usage. Also it helps to separate different qualities from each other, as they are influencing each other in different ways. Finally, the profiles can be used for most forms of assessment, including simulation.

Too large assessment efforts

For each of the driving quality requirements of the dialysis system architecture, research communities exist that have developed detailed assessment and evaluation methods for their quality attribute. In our experience, these techniques suffer from three major problems in the context of architecture assessment. First, they focus on a single quality attribute and ignore other, equally important, attributes. Second, they tend to be very detailed and elaborate in the analysis, requiring, sometimes, excessive amounts of time to perform a complete analysis. And finally the techniques are generally intended for the later design phases and often require detailed information not yet available during architecture design, e.g. source metrics like average depth of inheritance trees.

Since software architects generally have to balance a set of quality requirements, lack the data required by the aforementioned techniques and work under time pressure, the result is that, during architectural design, assessment is performed in an ad-hoc, intuition-based manner, without support from more formal techniques. Although some work e.g., (Kazman *et al.* 98), is performed in this area, there still is a considerable need for easy to use architecture assessment techniques for the various quality attributes, preferably with (integrated) tool support.

Architecture abstractions outside application domain

Traditional object oriented design methods (Booch, 1994; Jacobson *et al.*, 1992; Rumbaugh *et al.*, 1991; Wirfs-Brock *et al.*, 1990) provide hints and guidelines for finding the appropriate abstractions for the object oriented design. A common guideline is to take the significant concepts from the problem domain and objectify them, i.e. make them classes or

objects. However, in this project we observed that several of the architectural abstractions used in the final version did not exist (directly) in the application domain. Instead, these abstractions emerged during the design iterations and represented abstract domain functionality organized to optimize the driving quality attributes. We call these evolved abstractions *archetypes*.

Archetypes are not the same as design-patterns. Archetypes, as opposed to patterns, may not have been tried successfully before, nor is it described in a specific format. Archetypes may very well be good candidates for design patterns. Our definition and usage of the term ‘archetype’ differs from Shlaer and Mellor’s (1997).

We experienced that when a deeper understanding of the concept and its relations emerges, we found the most suitable abstraction. For example, during the first design iteration, we used the domain concepts we had learned from studying the documentation and talking to domain experts. As we came to know the requirements and the expected behavior of the system, the abstractions in the architecture changed from domain concepts to more abstract concepts and more easily met the quality requirements. During the design iterations, we became more and more aware of how the quality requirements would have to work in *cooperation*. For example, even though using design patterns might help with flexibility in some cases, the demonstrability and real-timeliness became hard to ensure and thus other abstractions had to be found.

Architecting is iterative

After the design of the dialysis system architecture we have come to the conclusion that designing architectures is necessarily an iterative activity and that it is impossible to get it completely right the first time. We designed the software architecture in two types of activities, i.e. individual design and group meeting design. We learned that group meetings and design teams meeting for two-three hours were extremely efficient compared to merging single individuals designs. Although one or two were responsible for putting things on paper and dealing with the details, virtually all creative design and redesign work was performed during these meetings.

In the case where one individual would work alone on the architecture it was very easy to get stuck with one or more problems. The problems were, in almost every case, resolved

the next design meeting. We believe that the major reason for this phenomenon is that the architecture design activity requires the architect to have almost all requirements, functional and quality, in mind at the same time. The design group has a better chance in managing the volume and still come up with creative solutions. In the group, it is possible for a person to disregard certain aspects to find new solutions. The other group members will ensure that those aspects are not forgotten.

Another problem we quickly discovered was that design decisions were forgotten or hard to remember between the meetings. We started early with writing design notes. The notes were very short, a few lines, with sketches where helpful. First, it helped us to understand why changes were made from previous design meetings. Secondly, it also made it easier to put together the rationale section of the architecture documentation. At some points, undocumented design decisions were questioned at a later stage and it took quite some time to reconstruct the original decision process.

The design notes we used were not exposed to inspections, configuration management or other formalizing activities. As such an informal document it was easy to write during the meeting. In fact, the designers soon learned to stop and have things written down for later reference. Since the sole purpose is to support the memory of the designers, often a date and numbering of the notes is enough.

Design aesthetics

The design activity is equally much a search for an *aesthetically appealing design* as it is searching and evaluating the balance of software qualities. The *feeling* of a good design worked as a good indicator when alternatives were compared and design decisions needed to be made. In addition, the feeling of disliking an architecture design often sparked a more thorough analysis and evaluation to find what was wrong. Most often, the notion proved correct and the analysis showed weaknesses.

According to our experience, the sense of a aesthetic design was often shared within the group. When differences in opinions existed, the problem or strength could be explained using a formal framework and we reached consensus. It is our belief that a software designer with roughly the same amount of

experience outside the project would experience the same feeling of aesthetic design. That is, although the “feeling” is not objective, it is at least intersubjective.

Since this intuitive and creative aspect of architecture design triggered much of the formal activities and analyses, we recognize it as very important. However, design methods, techniques and processes do not mention nor provide this as a part. It is not a secret but nor is it articulated very often how important this sense of design aesthetics, or gut feeling, is to software design.

Are we done?

We found it hard to decide when the design of the software architecture had reached its end criteria. One important reason is that software engineers are generally interested in technically perfect solutions and that each design is approaching perfectness asymptotically, but never reaches it completely. Architecture design requires balancing requirements and, in the compromise, requirements generally need to be weakened. Even if a first solution is found, one continues to search for solutions that require less weakening of requirements. Also, we found it very hard to decide when the architecture design was not architecture design anymore but had turned into detailed design.

A second important reason making it hard to decide whether a design is finished is that a detailed evaluation giving sufficient insight in the attributes of an architecture design is expensive, consuming considerable time and resources. Engineers tend to delay the detailed evaluation until it is rather certain that the architecture fulfils its requirements. Often, the design has passed that point considerably earlier. As we identified in the Introduction of this chapter, there is a considerable need for inexpensive evaluation and assessment techniques, preferably with tool support.

Documenting the essence of a software architecture

During the architecture design only rudimentary documentation was done, i.e. sketchy class diagrams and design notes. When we delivered the architecture to detailed design it had to be more complete. We tried to use the 4+1 View Model (Kruchten, 1995), but found it hard to capture the essence of

the architecture. Project members that had not participated in the design of the new architecture had to read the documentation and try to reconstruct this essence themselves. We have not yet been able to understand what the essence of a software architecture are, but we feel that its not equivalent with design rationale.

However, since we were able to communicate with the designers and implementers, we could overcome the problems with the documentation. The problem was put on its edge, when we started writing the conference paper to report on our findings. That time, we would not get a chance to communicate the architecture and its essence with any other means than this document. It is our opinion that although many of the aspects of this architecture are presented in this chapter, the essence still remain undocumented.

Architecture Description

The haemo dialysis architecture project started out with a very informal description of the legacy architecture, conveyed both in text and figures and via several discussions during our design meetings. For describing the resulting architecture we use two of the four views from the 4+1 View Model (Kruchten, 1995), i.e. Logical View and Process View. The development view we omit since it do not contribute to the understanding of the design decisions, trade offs and experiences. We also omit the physical view since the hardware is basically a single processor system. However, we feel that it is appropriate to add another subsection of our architecture description; the archetypes. During the design iterations we focused on finding suitable archetypes which allowed us to easily model the haemo dialysis architecture and its variants. The archetypes are very central to the design and important for understanding the haemo dialysis application architecture.

Logic Archetypes

When we started the re-design of the software architecture for the haemo dialysis machine, we were very much concerned with two things; the maintainability and the demonstrability.

We knew that the system had to be maintained by others than us, which meant that the archetypes we used, would have to

make sense to them and that the form rules were not limiting and easy to comprehend. Also, we figured, that if we could choose archetypes such that the system was easy to comprehend the effort to show what the system does becomes smaller. We realized that much of the changes would come from the MMI and new treatments and we needed the specification and implementation of a treatment to be easy and direct. Our aim was to make the implementation of the treatments look comparable to the description of a treatment written on a piece of paper by a domain expert using his or hers own terminology.

After three major iterations we decided on the Device/Control abstraction, which contained the following archetypes and their relations (figure 19):

- Device**

The system is modeled as a device hierarchy, starting with the entities close to the hardware up to the complete system. For every device, there are zero or more sub-devices and a controlling algorithm. The device is either a leaf device or a logical device. A leaf *device* is parameterized with a *controlling algorithm* and a *normalizer*. A logical device is, in addition to the *controlling algorithm* and the *normalizer*, parameterized with one or more sub devices.
- Controlling Algorithm**

In the device archetype, information about relations and configuration is stored. Computation is done in a separate archetype, which is used to parameterize Device components. The ControllingAlgorithm performs calculations for setting the values of sub output devices based on the values it gets from input sub devices and the control it receives from the encapsulating device. When the device is a leaf node the calculation is normally void.
- Normaliser**

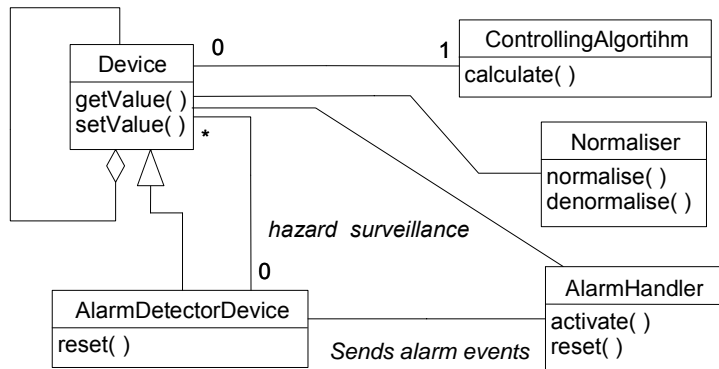
To deal with different units of measurement a normalization archetype is used. The normalizer is used to parameterize the device components and is invoked when normalizing from and to the units used by up-hierarchy devices and the controlling algorithm of the device.
- Alarm Detector Device**

Is a specialization of the Device archetype. Components of the AlarmDetectorDevice archetype is responsible for monitoring the sub devices and make sure the values read from the sensors are within the alarm threshold value set to the AlarmDetectorDevice. When threshold limits are crossed an AlarmHandler component is invoked.

Alarm Handler

The AlarmHandler is the archetype responsible for responding to alarms by returning the haemo dialysis machine to a safe-state or by addressing the cause of the alarm. Components are used to parameterize the AlarmDetectorDevice components

Figure 19.
The relations of the archetypes



Scheduling Archetypes

Haemo dialysis machines are required to operate in real time. However, haemo dialysis is a slow process that makes the deadline requirements on the system less tough to adhere to. A treatment typically takes a few hours and during that time the system is normally stable. The tough requirements in response time appear in hazard situations where the system is supposed to detect and eliminate any hazard swiftly. The actual timings are presented in medical equipment standards with special demands for haemo dialysis machines (CEI/IEC 601-2). Since the timing requirements are not that tight we designed the concurrency using the *Periodic Object pattern* (Molin and Ohlsson, 1998). It has been used successfully in earlier embedded software projects.

Scheduler

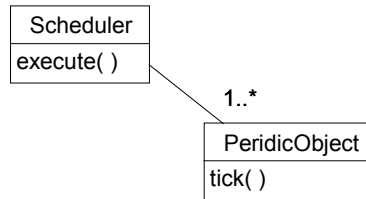
The scheduler archetype is responsible for scheduling and invoking the periodic objects. Only one scheduler element in the application may exist and it handles all periodic objects of the architecture. The scheduler accepts registrations from periodic objects and then distributes the execution between all the registered periodic objects. This kind of scheduling is not pre-emptive and requires usage of non-blocking I/O.

Periodic object

A periodic object is responsible for implementing its task using non-blocking I/O and using only the established time quanta. The *tick()* method will run to its completion and invoke the necessary methods to complete its task. Several periodic objects

may exist in the application architecture and the periodic object is responsible for registering itself with the scheduler (figure 20).

Figure 20.
*Basic Concurrency
with Periodic
Objects*



Connector Archetypes

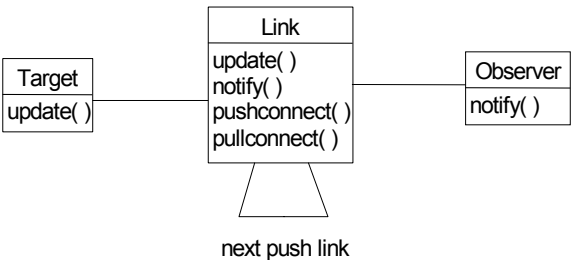
The communication between the architecture elements is done by using *causal connections* (Lundberg and Bosch, 1997). The principle is similar to the *Observer pattern* (Gamma *et al.*, 1995) and the *Publisher-Subscriber pattern* (Buschmann *et al.*, 1996). An observer observes a target but the difference is that a master, i.e. the entity registering and controlling the dependency, establishes the connection. Two different ways of communication exist, the push connection and the pull connection. In the first case, the target is responsible for the notifying the observer by sending the notify message. In the second case it is the observer that request data from the target. The usage of the connection allows for dynamic reconfiguration of the connection, i.e. push or pull. (figure 21)

Target The target holds data other entities are dependent on. The target is responsible for notifying the link when its state changes.

Observer The observer depends on the data or change of data in the target. Is either updated by a change or by own request.

Link The link maintains the dependencies between the target and its observers. Also holds the information about the type of connection, i.e. push or pull. It would be possible to extend the connection model with periodic updates.

Figure 21.
*Push/Pull Update
Connection*



Application Architecture

The archetypes represent the building blocks that we may use to model the application architecture of a haemo dialysis machine. In figure 23 the application architecture is presented. The archetypes allow for the application architecture to be specified in a hierarchical way, with the alarm devices being orthogonal to the control systems device hierarchy.

This also allows for a layered *view* of the system. For example, to specify a treatment we only have to interface the closest layer of devices to the HaemoDialysisMachine device (figure 23). There would be no need to understand or interfacing the lowest layer. The specification of a treatment could look like the example in figure 24.

Process View

**The Control
System**

The application architecture will be executed in pseudo parallel, using the periodic object pattern. In figure 22, the message sequence of the execution of one *tick()* on a device is presented. First, the Device collects the data, normalizes it using the normalizer parameter and then calculates the new set values using the control algorithm parameter.

Figure 22.
*The message
sequence of a control
tick()*

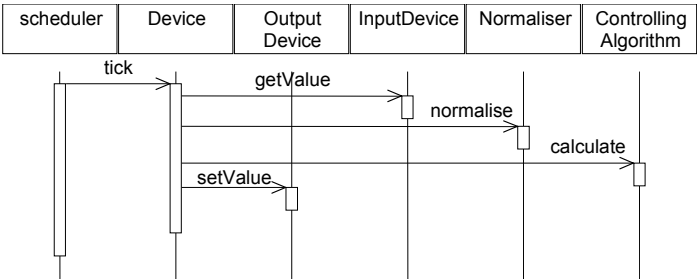


Figure 23.
*Example haemo
dialysis Application
Architecture*

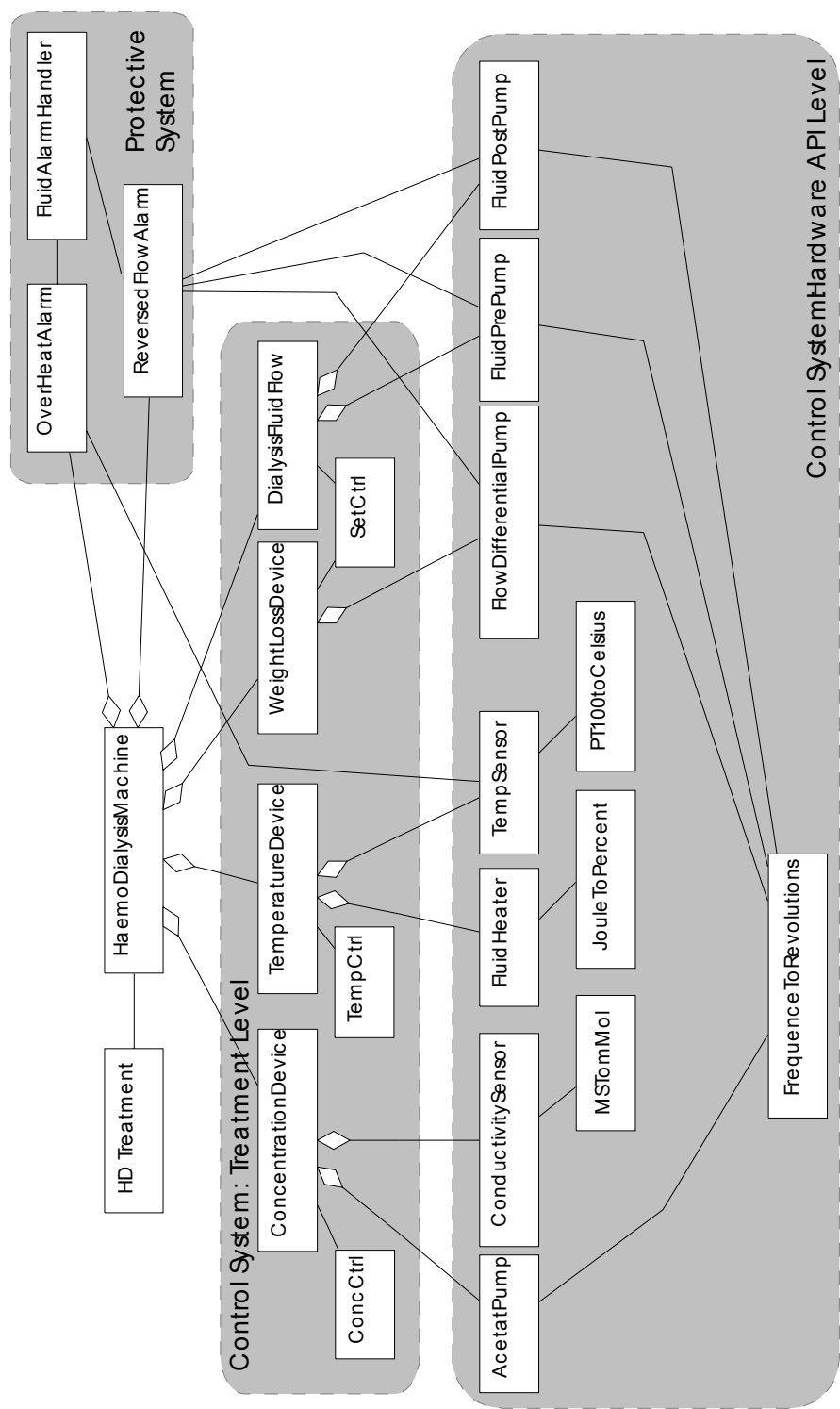


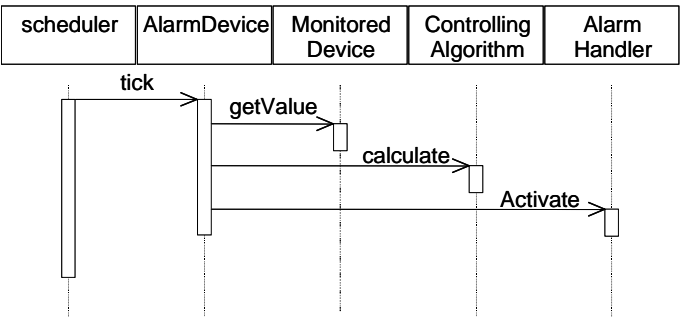
Figure 24.
*Example of
treatment code
specification*

```
conductivity.set(0.2); // in milliMol
temperature.set(37.5); // in Celsius
weightloss.set(2000); // in milliLitre
dialysisFluidFlow.set(200); // in milliLitre per
minute
overHeatAlarm.set(37.5,5); // ideal value in
// Celsius and maximum deviation in percent
treatmentTimer.set(180); // in minutes
start(); // commence the treatment
```

**Alarm
Monitoring**

The control system may utilize AlarmDevices to detect problem situations and the protective system will consist of a more complex configuration of different types of AlarmDevices. These will also be run periodically and in pseudo parallel. The message sequence of one *tick()* for alarm monitoring is shown in figure 25.

Figure 25.
*A tick() of alarm
monitoring*



**Treatment
Process**

The treatment process is central to the haemo dialysis machine and its software. The general process of a treatment consists of the following steps:

- 1 preparation,
- 2 self test,
- 3 priming,
- 4 connect patient blood flow,
- 5 set treatment parameters,
- 6 start dialysis,
- 7 treatment done,
- 8 indication,
- 9 nurse feeds back blood to patient,

- 10 disconnect patient from machine,
- 11 treatment records saved,
- 12 disinfecting,
- 13 power down.

The process generally takes several hours and the major part of the time the treatment process are involved in a monitoring and controlling cycle. The more detailed specification of this sub process is here.

- 1 Start fluid control
- 2 Start blood control
- 3 Start measuring treatment parameters, e.g. duration, weight loss, trans-membrane pressure, etc.
- 4 Start protective system
- 5 Control and monitor blood and fluid systems until time-out or accumulated weight loss reached desired values
- 6 Treatment shutdown

Rationale

During the design of the haemo dialysis architecture we had to make a number of design decisions. In this section the major design decisions and their rationale are presented.

The starting point

From the start we had a few documents describing the domain and the typical domain requirements. In the documents, the subsystems of the old system were described. Also, we had earlier experiences from designing architectures for embedded systems, i.e. Fire Alarm Systems (Molin and Ohlsson, 1998) and Measurement Systems (Bosch, 1999a). We started out using the main archetypes from these experiences which were *sensors* and *actuators*.

Initially, we wanted to address the demonstrability issues by ensuring that an architecture was easy to comprehend, consequently improving maintainability. The intention was to design an architecture that facilitated visualization and control of the functions implemented in the final application. Especially important is demonstration of the patient safety during treatments.

Our goal was to make the specification and implementation of the treatments very concise and to as high extent as possible look like the specification of a treatment that a domain expert would give, i.e. using the same concepts, units, and style.

The result was that our initially chosen abstraction, *sensor* and *actuators* did not suit our purpose adequately. The reason is that the abstraction gives no possibility of shielding the hardware and low-level specifics from the higher-level treatment specifications.

The iterations

The architecture design was iterated and evaluated some three times more, each addressing the requirements of the previous design and incorporating more of the full requirement specification.

In the first iteration, we used the Facade design pattern (Gamma *et al.*, 1995) to remedy the problem of hiding details from the treatment specifications. Spurred by the wonderful pattern we introduced several facades in the architecture. The result was unnecessary complexity and did not give the simple specification of a treatment that we desired.

In the second iteration, we reduced the number of facades and adjusted the abstraction, into a *device hierarchy*. This allowed us to use sub-devices that were communicating with the hardware and dealt with the low-level problems such as normalization and hardware APIs. These low-level devices were connected as logical inputs and outputs to other logical devices. The logical devices handle logical entities, e.g. a heater device and a thermometer device are connected to the logical device Temperature (figure 23). This allows for specification of treatments using the vocabulary of the logical devices, adapted from the low level hardware parameters to the domain vocabulary.

In the third major iteration, the architecture was improved for flexibility and reuse by introducing parameterization for normalization and control algorithms. Also the alarm detection device was introduced for detecting anomalies and hazards situations.

Concurrency

The control system involves constantly executing control loops that evaluate the current state of the process and calculates new set values to keep the process at its optimal parameters. This is supposed to be done simultaneously, i.e. in parallel. However,

the system is in its basic version only equipped with a single processor reducing parallelism to pseudo parallel. On a single processor system we have the options of (1) choose to use a third party real-time kernel supporting multi-threads and real-time scheduling. And (2) we can design and implement the system to be semi-concurrent using the periodic objects approach and make sure that the alarm functions are given the due priority for achieving swift detection and elimination of hazards. Finally (3) we may choose the optimistic approach, i.e., design a sequentially executing system and make it fast enough to achieve the tightest response time requirements.

The first one is undesirable because of two reasons, i.e. resource consumption and price. The resources, i.e. memory and processor capacity, consumed by such a real-time kernel are substantial especially since we most likely will have to sacrifice resources, e.g. processor capacity and memory, for services we will not use. In addition, the price for a certified real-time kernel is high and the production and maintenance departments become dependent on third-party software.

The third option is perhaps the most straightforward option and could probably be completed. However, such a strategy may affect the demonstrability negatively. Because of the software certification, it is unrealistic to believe that such an implementation would be allowed in a dialysis machines.

The second option, pose limitations in the implementation and design of the system, i.e. all objects must implement their methods using non-blocking I/O. However, it still is the most promising solution. Periodic objects visualize the parallel behavior clearly, using the scheduler and its list of periodic objects especially since it has been used successfully in other systems.

Communication The traditional component communication semantics are that a sender sends a message to a known receiver. However, this simple message send may represent many different relations between components. In the design of the dialysis system architecture, we ran into a problem related to message passing in a number of places. The problem was that, in the situation where two components had some relation, it was not clear which of the two components would call the other component. For example, one can use a *pushing* approach, i.e. the data generating component pushing it to the interested parties, or a

pulling approach, where the interested components inquire at the data generating component, and each approach requires a considerable different implementation in the involved components.

As a solution, the notion of *causal connections* (Lundberg and Bosch, 1997) was introduced that factors out the responsibility of abstracting the calling direction between the two components such that neither of the components needs to be concerned with this.

The advantage of using a causal connection is that the involved components can be focused on their domain functionality and need not concern about how to inform or inquire the outside world, thus improving the reusability of the components. In addition, it allows one to replace the type of causal connection at run-time, which allows a system to adapt itself to a changing context during operation.

Evaluation

In this section an analysis of the architecture design is presented with respect to the quality requirements. As stated in section (Too large assessment efforts), the traditional assessment methods are inappropriate for the architecture level and therefore our evaluation was strongest on maintainability (prototype) and more subjective for the other quality requirements.

Maintainability

To evaluate the maintainability and feasibility of the architecture the industrial partner EC-Gruppen developed a prototype of the fluid-system. The prototype included controlling fluid pumps and the conductivity sensors. In total the source code volume for the prototype was 5,5 kLOC.

The maintainability was validated by an extension of the prototype. Changing the pump process control algorithms, a typically common maintenance task. The change required about seven (7) lines of code to change in two (2) classes. And the prototype was operational again after less than two days work from one person. Although this is not scientifically valid evidence, it indicates that the architecture easily incorporates the expected types of changes.

Reusability

The reusability of components and applications developed using this architecture has not been measured, for obvious reasons. But our preliminary assessment shows that the sub quality factors of reusability (McCall, 1994), i.e. generality, modularity, software system independence, machine independence and self-descriptiveness, all are reasonably accounted for in this architecture. First, the architecture supports generality. The device archetype allow for separation between devices and most of the application architecture will be made of devices of different forms. Second, the modularity is high. The archetypes allows for clear and distinguishable separation of features into their own device entity. Third, the architecture has no excessive dependencies to any other software system, e.g. multi processing kernel. Fourth, the hardware dependencies have been separated into their own device entities and can easily by substituted for other brands or models. Finally, the archetypes provide comprehensible abstraction for modeling a haemo dialysis machine. Locating, understanding and modifying existing behavior is, due to the architecture, an easy and comprehensible task.

Safety

The alarm devices ensure the safety of the patient in a structured and comprehensible way. Every hazard condition is monitored and has its own AlarmDetectorDevice. This makes it easier to demonstrate what safety precautions have been implemented from the standard.

Real-timeliness

This requirement was not explicitly evaluated during the project. Instead our assumption was that the data processing performance would equal that of a Pentium processor. Given that the prototype would work on a Pentium PC running NT, it would be able to run fast enough with a less resource consuming operating system in the haemo dialysis machine.

Demonstrability

Our goal when concerned with the demonstrability was to achieve a design that made the specification of a treatment and its complex subprocesses very similar to how domain experts would express the treatment in their own vocabulary. The

source code example in the Architecture section for specifying a treatment in the application is very intuitive compared to specifying the parameters of the complex sub processes of the treatments. Hence, we consider that design goal achieved.

Conclusions

In this chapter, the architectural design of a haemo dialysis system and the lessons learned from the process leading to the architecture have been presented. The main experiences from the project are the following:

First, quality requirements are often specified without any context and this complicates the evaluation of the architecture for these attributes and the balancing of quality attributes.

Second, assessment techniques developed by the various research communities studying a single quality attribute, e.g. performance or reusability, are generally intended for later phases in development and require sometimes excessive effort and data not available during architecture design.

Third, the archetypes use as the foundation of a software architecture cannot be deduced from the application domain through domain analysis. Instead, the archetypes represent chunks of domain functionality optimized for the driving quality requirements.

Fourth, during the design process we learned that design is inherently iterative, that group design meetings are far more effective than individual architects and that documenting design decisions is very important in order to capture the design rationale.

Fifth, architecture designs have an associated aesthetics that, at least, is perceived inter-subjectively and an intuitively appealing design proved to be an excellent indicator.

Sixth, it proved to be hard to decide when one was done with the architectural design due to the natural tendency of software engineers to perfect solutions and to the required effort of architecture assessment.

Finally, it is very hard to document all relevant aspects of a software architecture. The architecture design presented in the previous section provides some background to our experiences.

CHAPTER 7: **Beer-Can Inspection Case**

This chapter presents the case study in which we applied the architecture level modifiability analysis method (ALMA) to compare the software architecture between design iterations of a beer-can inspection system. The beer can system is a research system developed as part of a joint research project between the company EC-Gruppen AB and the software architecture research group at the Blekinge Institute of Technology. The goal of the project was to define a DSSA that provides a reusable and flexible basis for instantiating measurement systems. Although the software architecture of the beer can inspection system is a rather prototypical instance of a measurement system, we used it as a starting point.

A challenge in design projects is to decide when one has reached the point where the architecture fulfils its requirements. At this stage in development, the functional requirements can, in general, be evaluated relatively easy by tracing the requirements in the design. Software quality requirements such as reusability and robustness, on the other hand, are more difficult to assess.

The next sub-section presents the domain and domain requirements on measurement systems. The next section describes the analysis goal in the case study, followed by sections describing the software architecture, the scenario elicitation, the scenario evaluations for each iteration and the interpretation of those results. The final section in this chapter presents the conclusions from this case.

Automatic Beer-Can Inspection

The automatic inspection system is an embedded system located at the beginning of a beer can filling process and its goal is to remove dirty beer cans from the input stream. Clean cans should just pass the system without any further action.

The system consists of a triggering sensor, a camera and an actuator that can remove cans from the conveyer belt. When the hardware trigger detects a can, it sends an trigger event to the software system. After a predefined amount of time, the camera takes a number of image samples of the can. Subsequently, the measured values, i.e., images, are compared to the ideal images and a decision about removing or not removing the can is made. If the can should be removed, the system invokes the actuator at a predefined time point relative to the trigger event. Figure 26 presents the process graphically.

Figure 26.
*Example beer cans
measurement system*

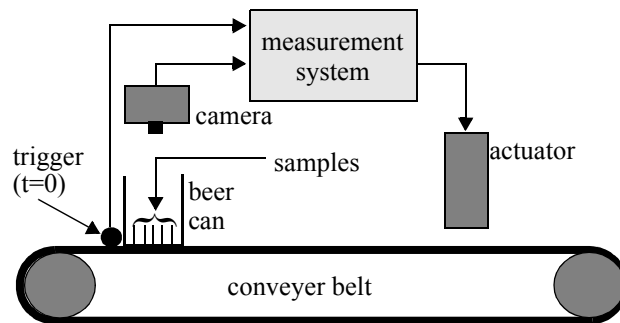
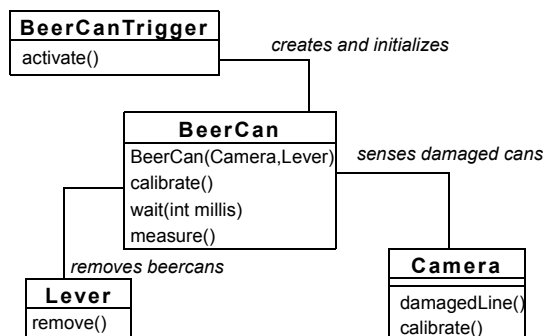


Figure 27 presents the application architecture of the beer can system. The architecture was designed based on the functional requirements of the system without any explicit design effort with respect to software quality requirements such as reusability and performance.

Figure 27.
*Object model of the
beer can application*



We use the architecture of the presented system as the basis for creating a domain-specific software architecture (DSSA)

(D'Ippolito, 1990; Tracz, 1995) that allows the software engineer to instantiate applications in the domain of measurement systems.

Measurement Systems

The domain of measurement systems denotes a class of systems used to measure the relevant values of a process or product. These systems are different from the, better known, process control systems in that the measured values are not directly, i.e., as part of the same system, used to control the production process that creates the measured product or process. Industry uses measurement systems for quality control on produced products, e.g., to separate acceptable from unacceptable products or to categorise the products in quality grades.

Goal Setting

The goal with the analysis was to compare two alternative architecture designs, namely the baseline version of the revised version. In this section we illustrate how to use ALMA to compare the new version of the architecture with the old to confirm that the new version was indeed an improvement over the previous version. In total we made six design iterations each with an associated analysis. In comparing candidates we may of course choose predict modifiability for each candidate and compare the results, or perform a risk assessment and compare the results.

Architecture Description

To compare two software architecture candidates we need descriptions of both candidates. In this case study we had already the logical view (Kruchten 1995) that described the systems' decomposition and the components' relations using very basic UML-like notation, illustrated in figurea 28 and 29. The system is an embedded control system with no firm relations to other software systems and therefore there was no need to describe the systems environment in an architectural view. Although the logical view was described rather rudimentarily, it did not present any major problem, since we, the assessors, were also part of the design process we also had the informal information available to us.

The comparison of two candidate architectures requires a single common frame of reference. When using scenarios for evaluation the frame of reference is the set of change scenarios that is used to assess the modifiability of all the candidate architectures. The scenario set is part of the representation of the modifiability requirements that are the same for all candidates.

In change scenario elicitation for comparing candidate software architectures, the aim is to elicit scenarios that reveal the relevant differences between the candidates. Similar to the other two goals, we need to decide which stakeholders should contribute to the scenario list. In this case scenarios were based on information we received from our partner company.

The stakeholders are then interviewed and asked to enumerate the scenarios they consider critical or extreme for the system in relation to modifiability. The idea is to stress test the candidates. The contributors are then asked to review and revise the synthesized list in the same way as for the maintenance prediction.

Software quality requirements

The DSSA for measurement systems should fulfil a number of requirements. The most relevant software quality requirements in the context of this work are reusability and maintainability. Measurement systems also have to fulfil real-time and robustness requirements, but we leave these out of the discussion here.

Reusability

The reusability quality attribute provides a balance between the two properties; generality and specifics. First, the architecture and its components should be general because they should be applied in other similar situations. For example, a weight sensor component can be sufficiently generic to be used in several applications. Secondly, the architecture should provide concrete functionality that provides considerable benefit when it is reused.

To evaluate the existing application architecture we use scenarios. However, reusability is a difficult software property to assess. We evaluate by analysing the architecture with respect to each scenario and assess the ratio of components reused *as-is* and total number of components. These are the scenarios:

R1 Product packaging quality control. For example, sugar packages that are both measured with respect to intact packaging and weight.

R2 Surface finish quality control where multiple algorithms may be used to derive a quality figure to form a basis for decisions.

R3 Quality testing of microprocessors where each processor is either rejected or given a serial number and test data logged in a quality history database in another system.

R4 Product sorting and labelling, e.g., parts are sorted after tolerance levels and labelled in several tolerance categories and are sorted in different storage bins.

R5 Intelligent quality assurance system, e.g., printing quality assurance. The system detects problems with printed results and rejects occasional misprints, but several misprints in a sequence might cause rejection and raising an alarm.

All presented scenarios require behaviour not present in the initial software architecture. However, the scenarios are realistic because measurement systems exist that require functionality defined by the scenarios.

Maintainability

In software-intensive systems, maintainability is generally considered important. A measurement system is an embedded software system and its function is very dependent on its context and environment. Changes to that environment often inflict changes to the software system. The goal for maintainability in this context is that the most likely changes in requirements are incorporated in the software system against minimal effort.

In addition, maintainability of the DSSA is assessed using scenarios. For the discussion in this paper, the following scenarios are applied on the DSSA.

M1 The types of input or output devices used in the system is excluded from the suppliers assortment and need to be changed. The corresponding software needs to be updated.

M2 Advances in technology allows a more accurate or faster calculation to be used. The software needs to be modified to implement new calculation algorithms.

M3 The method for calibration is modified, e.g., from user

activation to automated intervals.

M4 The external systems interface for data exchange change. The interfacing system is updated and requires change.

M5 The hardware platform is updated, with new processor and I/O interface.

These are the scenarios we have found to be representative for the maintenance of *existing* measurement systems. Of course, other changes may possibly be required, but are less likely to appear.

Change Scenario Evaluation

The effect of each scenario in the profile is evaluated against all candidate architectures. The evaluation is done by applying the impact analysis described in “Change scenario evaluation” on page 59: identify the affected components, determine the effect on those components, and determine ripple effects. The effect is then expressed in a way that allows comparison between the candidates. We may choose to evaluate and express the results in one of the following ways:

- For each scenario, determine the candidate architecture that supports it best, or conclude that there are no differences. The results are expressed as a list of the scenarios with the best candidate for each scenario.
- For each scenario, rank the candidate architectures depending on their support for the scenario and summarize the ranks.
- For each scenario, determine the effect on the candidate architectures and express this effect using at least an ordinal scale., e.g. the estimated number of lines of code affected.

In the beer can inspection case we chose to express the impact as the component modification ratio, i.e. modified components divided by the total number of components. For each new version we evaluated the scenarios and compared to the previous version. Table 13 shows the initial result (iteration no. 0) , the results after each of the iterations and the final result.

Architecture evaluation

We analyse the application architecture by asking a typical question for reusability, i.e., *How much will I be able to reuse of the software*¹ in the context of each reuse scenario. The results from this analysis and the transformations are shown in table 13. Every scenario is assigned a ratio of affected components in the scenario divided by the total number of components in the current architecture. For reusability this should be as close to one as possible, i.e., as many of the components as possible should be reusable as-is. For maintainability, this should be as low as possible, i.e., as few components as possible should need to be modified. The remainder of this section presents the evaluation results.

Reusability Evaluation

Analysing the initial application using R1, we find that we can probably reuse the camera and the lever components. These are not necessarily dependent on the rest of the components to be useful. We get similar results from R2.

In R3, we find that the sensing device will have to be more complex and include sophisticated algorithms to work. The same goes for the actuating device that now also needs to be fed the data to imprint on the product. Therefore, it is most likely that none of the components can be reused. Similar for R4 we find increasingly complex actuation schemes. Hence, we cannot expect any reuse in this scenario either.

In R5 we find that the actuation device possibly could be reused. Because of using previous results, i.e., the actuation history, we need more sophisticated measurement items.

Maintainability Evaluation

Then we follow the same procedure with the maintenance scenarios. The question replaced, by *How easy is it to fix*² in the context of each maintenance scenario. These are the results:

In M1, we see that changing hardware interface for the lever, the trigger or the camera is possible to do without modification to any other components. It is concise enough. For one change only one component has to be modified.

It is worse for the result of M2. Modifying the calculation algorithm in the initial architecture is impossible for us to exclude changes to any of the components. Possibly, all

1. Modified from the original, 'Will I be able to reuse some of the software?' (McCall 94).

2. Modified from the original, 'Can I fix it?' (McCall 94).

components will have to be updated. Similar is the situation for M3. We cannot exclude any component that requires no modification.

In the case of M4, we may if the detailed design was done well enough get by with modification to only one component. Again, we cannot exclude any of the other components.

The last maintenance scenario, M5, will most likely require modification to all the components. No component can be excluded as not using any hardware specific services from the platform.

Based on the results from the analysis we make the following conclusions about the application architecture.

- Reusability is not to good in the initial architecture. The classes are tightly coupled and the reuse scenarios show limited possibilities to reuse as-is.
- Maintainability could also be better. One of the scenarios (M1) is satisfied. The other scenarios however, are not supported satisfactory. The main problem seems to be that a change is not very concise and not at all simple.

The results of the analysis indicate that the reusability and maintainability attributes of the architecture are not satisfying. In the next section, we present transformations for improving these attributes.

First Iteration - Component level transformations

Problem Although the beer cans application is a small set of classes, the task of changing or introducing a new type of items require the source code of most components to be changed. In all the specified reuse scenarios, the use of new types of measurement item is involved.

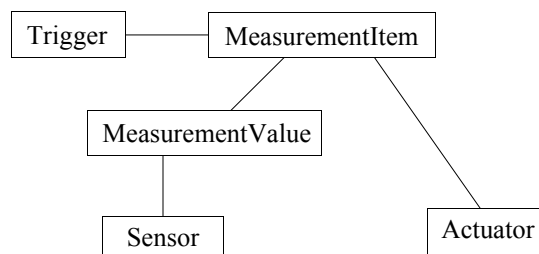
Alternatives We first look for an architectural style that might fit our needs. An alternative could be to organise the system into a pipe&filter architecture with sensing, data interpreting and actuation filters. A second alternative, and perhaps more natural for object oriented designers, is to capture the real-world correspondents of these classes and define relevant abstractions, i.e., to define them as components with an interface and behaviour. Support for this can be found in the scenarios.

Transformation The pipes & filters alternative requires changes to more than one filter for several scenarios, i.e., R2-R5 and M3-M5. Instead, we choose to ‘componentify’ the classes into the generic component definitions. The DSSA should be able to deal with the different kinds of items in a uniform way, and therefore the abstraction `MeasurementItem` is introduced.

Subsequently, component types `Actuator`, for the `RemoveLever`, and `Sensor`, for the `LineCamera`, are defined. Different types of actuators can be defined as components of the type `Actuator`. The `Trigger` is defined as a specialised sensor component, with interface additions. The redesigned architecture is presented in figure 28. In addition, the affected components are marked with (1) in figure 29.

Problem resolved The components introduced reduced the coupling between the components by use of the principle programming towards an interface and not an implementation. General software engineering experience tell us that this principle of programming give us benefits both in situations of reuse and maintenance.

Figure 28.
Model after first transformation



Second Iteration - Separation of component creation from trigger

Problem The introduction of the component types facilitates the introduction of several concrete components but does not remove the problem of type dependence at the time of component creation. Every time a client creates a component it needs to know the actual sensor, actuator and measurement item types. From a maintainability perspective, this is not at all concise or simple to modify.

Alternatives One can identify two alternatives to address this problem. The first is to introduce pre-processor macros in the source code to

easily change all instances of component creation of the affected types. The second alternative is to use the Abstract Factory (Gamma *et al.* 94) to centralise the information about concrete types.

Transformation There are a number of drawbacks with the macro alternative; for example, it is a static solution that when changed, it must be recompiled. The ItemFactory could have an interface for changing the instantiation scheme. Therefore the Abstract Factory pattern is selected and a factory component is introduced to handle the instantiation of actuators, sensors and measurement items. See (2) in figure 29.

Problem resolved The trigger need no longer to know the actual type of measurement item to be created, but instead requests a new measurement item from the ItemFactory. The use of the Abstract Factory pattern did eliminate that problem.

Third Iteration - Changing to strategies

Problem Different components in the measurement system perform similar tasks. Changes to these tasks require updating the source code of every component containing the same implementation for the task. For example, the measurement of a measurement item aspect could be performed by different methods. A change to the similar parts is not concise enough in the current design and inhibits the maintainability. The measurement item may pull data from sensor, the sensor may push data to measurement item or the sensors may pass on data whenever they are changed. These alternatives are common to all sensors, independent of their types, and a way of reusing this aspect of the sensor is desired.

Alternatives The Strategy pattern (Gamma *et al.* 94) allows more distinction between the method for getting input data and the method of deriving the actual value for the actuation decision.

Transformation This increases the conciseness and the modularity of the architecture and should improve the maintainability. Since the reuser selects one out of many strategies the reusability will be somewhat reduced as a smaller number of components is reused as is. However, the benefits for maintainability outweigh the liabilities with loss in reusability. The Strategy pattern is applied to all components where we can identify a similar need

- *Sensor update strategy.* The three variations (push, pull, on change) apply to how sensors pass their data. The strategies are OnChangeUpdate, ClientUpdate, and PeriodicUpdate.
- *Calculation strategy.* Calculation of derived data and decisions show that there are similar methods to perform different tasks of different objects. Different calculation strategies can be defined. These strategies can be used by sensors, measurement items and actuators.

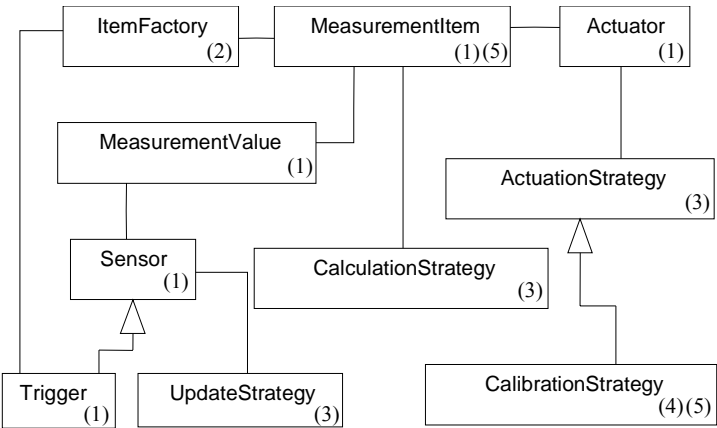
The introduced components are marked with (3) in figure 29.

Problem resolved Maintainability improved greatly from this transformation, to the cost of some reusability. The gain was substantial enough to motivate the trade-off.

Fourth Iteration - Unification of actuation and calibration

Problem Calibration is performed by letting the system measure a manually inspected item and the values are stored as ideal. In normal service, deviations from the measured ideal values are considered faults. The calibration is supposed to be carried out at different times in run-time. Calibration of the measurement system is considered a special service not implemented using the concepts of the current architecture. This makes the system more tightly coupled, in the sense that the factory, the measurement item and the sensors need to be aware of the calibration state. This makes several components dependent of a common state and changes to modify calibration becomes complex. This is not good for maintainability, or for reusability.

Figure 29.
The Measurement Systems DSSA¹



Alternatives Two alternatives to decreasing the coupling due to dependency of the common state exist. First, the calibration is defined as a separate service that all sensors and measurement items must implement. Each component checks if its current state is calibration and then call its calibration services instead of the normal services. The second alternative is to use the strategy design pattern and identify that most of the calibration is really the same as for the normal measuring procedure. The difference is that the ideal measures have to be stored for later referencing.

Transformation The first alternative is not desirable since this introduces behaviour that is not really the components responsibility. Instead we favour the second alternative and introduce a special case of an actuation strategy, i.e., the CalibrationStrategy. The result is that the calibration is performed with the calibration actuation strategy, and when invoked stores the ideal values where desired. See (4) in figure 29.

Problem resolved The use of the calibration strategy as a special type of actuation removes the dependence on a global state. This reduces the need to modify several components when modifying the calibration behaviour. Consequently, we have improved the maintainability.

Fifth Iteration - Adding prototype to factory

Problem The introduction of the calibration strategy addressed most of the identified problems. However, there is a problem remaining, i.e., the calibration strategy is dependent on the implementation type the measurement item component. This couples these two, so that the calibration strategy cannot be reused separate from the measurement item.

Alternatives One alternative solution is to decrease the coupling by introducing intermediate data storage, e.g., a configuration file, where the ideal values are stored. The second alternative is to apply the Prototype design pattern (Gamma *et al.* 94). The ideal measurement item is stored in the ItemFactory and used as a template for subsequent measurement item instances.

Transformation We decide to apply a variant of the prototype pattern. The calibration of the system is performed by setting the ItemFactory into calibration mode. When a real-world item triggers the ItemFactory, it creates a new measurement item with the normal sensors associated. The measurement item collects the data by

reading the sensors, but when invoking the actuation strategy, instead of invoking the actuators, the calibration strategy causes the measurement item to store itself as a prototype entity at the item factory. The affected relation is marked (5) in figure 29.

Problem resolved The Prototype pattern was applied to the ItemFactory. Consequently, the calibration strategy no longer needs to know the concrete implementation of the measurement item. The only component in the system with knowledge about concrete component types is the ItemFactory. The information has been localised in a single entity.

Final assessment

In table 13, the results from the final assessment of the domain specific software architecture for measurement systems are presented. Figure 29 contains the most relevant classes of the DSSA. As shown in the evaluation, the scenario M5 is not supported by the DSSA. However, since we estimate the likelihood for scenario M5 rather low, we accept the bad score for that scenario. The results of the second transformation are especially relevant, since it illustrates the trade-off between reusability and maintainability (see figure 30). Overall, we find the result from the transformations satisfying and the analysis of the scenarios shows substantial improvement.

Figure 30.
The effect of each transformation

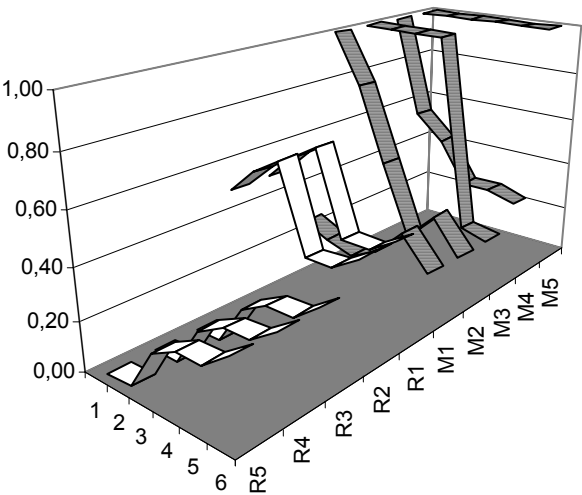


Table 13: Analysis of architecture

Software Quality		Iteration no.					
	Scenario	0	1	2	3	4	5
Reusability	R1	2/4	3/5	4/6	3/9	3/9	4/10
	R2	2/4	3/5	4/6	3/9	3/9	4/10
	R3	0/4	0/5	1/6	2/9	2/9	3/10
	R4	0/4	0/5	1/6	2/9	2/9	3/10
	R5	0/4	0/5	1/6	2/9	2/9	3/10
Maintainability	M1	1/4	1/5	1/6	2/9	3/9	2/10
	M2	4/4	4/5	3/6	2/9	3/9	2/10
	M3	4/4	5/5	6/6	9/9	2/9	2/10
	M4	4/4	3/5	3/6	3/9	3/9	3/10
	M5	4/4	5/5	6/6	9/9	9/9	10/10

Interpretation

When the goal of the analysis is to compare candidate architectures, the interpretation is aimed at selecting the best candidate. In section “Change Scenario Evaluation” on page 140 we distinguished three approaches to compare candidate architectures: (1) appoint the best candidate for each scenario, (2) rank the candidates for each scenario, and (3) estimate the effort of each scenario for all candidates on some scale.

When using the first approach, the results are interpreted such that the architecture candidate that supports most scenarios is considered best. In some cases we are unable to decide on the candidate that supports a change scenario best, for instance because they require the same amount of effort. These scenarios do not help discriminate between the candidates and do not require any attention in the interpretation. This interpretation of the results allows the analyst to understand the differences between the candidate architectures in terms of effects of concrete changes.

When we rank the candidates for each scenario, we get an overview of the differences in the group of candidate

architectures. Based on some selection criterion, we then select the candidate architecture that is most suitable for the system. For instance, we may select the candidate that is considered best for most scenarios. Or, alternatively, we may choose the candidate that is never considered worst in any scenario. The stakeholders, together with the analyst, decide on the selection criterion to be used.

When we use the third approach, the interpretation can be done in two ways: comprehensive or aggregated. Using the first interpretation, the analyst considers the results of all scenarios to select a candidate. For instance, the analyst could, based on all the scenarios, decide that certain scenarios are the most important scenarios to see which candidate architecture is best. This type of interpretation relies on the experience and analytical skills of the analyst. The other approach is to aggregate the results for all candidate architectures using some kind of algorithm or formula, such as the prediction model on page 159. The predicted effort is then compared and the best candidate has the lowest predicted effort. This interpretation focuses on the whole rather than the differences. These two interpretation techniques can be used together to come to a better judgment, because they are based on the same results.

In the beer can inspection case we chose the more comprehensive approach to interpreting the results. By doing so we get more insights as to what problems to be addressed in the next iteration of the design. In table 13 we see that the architecture was improved in terms of component modification ratio for the main part of the scenarios and remained on the same level for two of the change scenarios. Figure 30 illustrates the improvements of the architecture during the iterations. This figure summarizes the results of all six analyses for each change scenario (S1 to S5) made during the design iterations. We can clearly see that for change scenario 1 (S1) and change scenario 2 (S2), the fourth iteration worsened the results. However, the same iteration greatly improved the results for change scenario 3 (S3). This is a clear trade-off within a quality aspect. Because it is made explicit the architect may consciously decide whether it is acceptable, or not.

In this section we have presented a case study where we used and early version of the architecture level modifiability analysis method (ALMA) to compare candidate software architectures in a iterative design process. The technique is based on finding extreme scenarios to stress the architecture and expose differences. Evaluation for this goal can be performed in slightly different ways: determine the best candidate for each scenario, rank the scenario, or express the effect of the scenario on an ordinal scale for both candidates. Interpretation of the results is dependent on the evaluation scheme used. It may determine what candidate is best for most scenarios, or summarize the ranks and compare the sum, or compare the quantitative results.

In the beer can inspection case we concluded from the assessment by comparing the ratio of required modifications that the design iterations had on total improved the architecture regarding modifiability, i.e the last version required less components to change. Although the path, as presented in figure 30, shows that some steps did not improve the outcome for all scenarios, the end result is indeed an improvement. Worth noting is that the fifth scenario did not improve at all during the iterations and had, when concerned with this comparison only, no contribution to the result.

Iterative architecture design with architecture analysis at each iteration provide an objective way to gradually improve the design. Associated with each iteration, one or more problems are explicitly addressed, alternative solutions are investigated and the rationale for the design decisions is captured. Together with the almost immediate feed-back, the architect quickly learns where the rationale and solutions strategies holds.

CHAPTER 8: **Mobile Positioning Case**

This chapter presents the case study in which we used the architecture level modifiability analysis method (ALMA) for predicting modification effort of the Mobile Positioning Center (MPC) at Ericsson Software Technology AB, a Swedish Ericsson subsidiary. The goal with the case study was to make a prediction of the effort required to modify the system in later releases. The challenge in this case study was that we could not involve all stakeholders.

The next sub-section briefly describes the mobile positioning domain. The remainder of the chapter is organized after the steps of the analysis method, starting with a section on the goal setting, followed by sections describing the software architecture, the scenario elicitation, the scenario evaluation and the results interpretation. The last section in this chapter presents the conclusions.

Mobile Positioning

The MPC is a system for locating mobile phones in a cellular network and reporting their geographical position. The MPC extends a telecom network with a mobile device positioning service, and enables network operators to implement services using the positioning information provided by the MPC.

The system consists of the MPC server and a Graphical User Interface (GUI) as a client. The MPC server handles all communication with external systems to retrieve positioning data and is also responsible for the processing of positioning data. The MPC server also generates billing information that allows network operators to charge the customers for the services they use. The MPC GUI is used for system administration, such as configuring for which mobile phones a user is allowed to retrieve the position and configuring alarm handling.

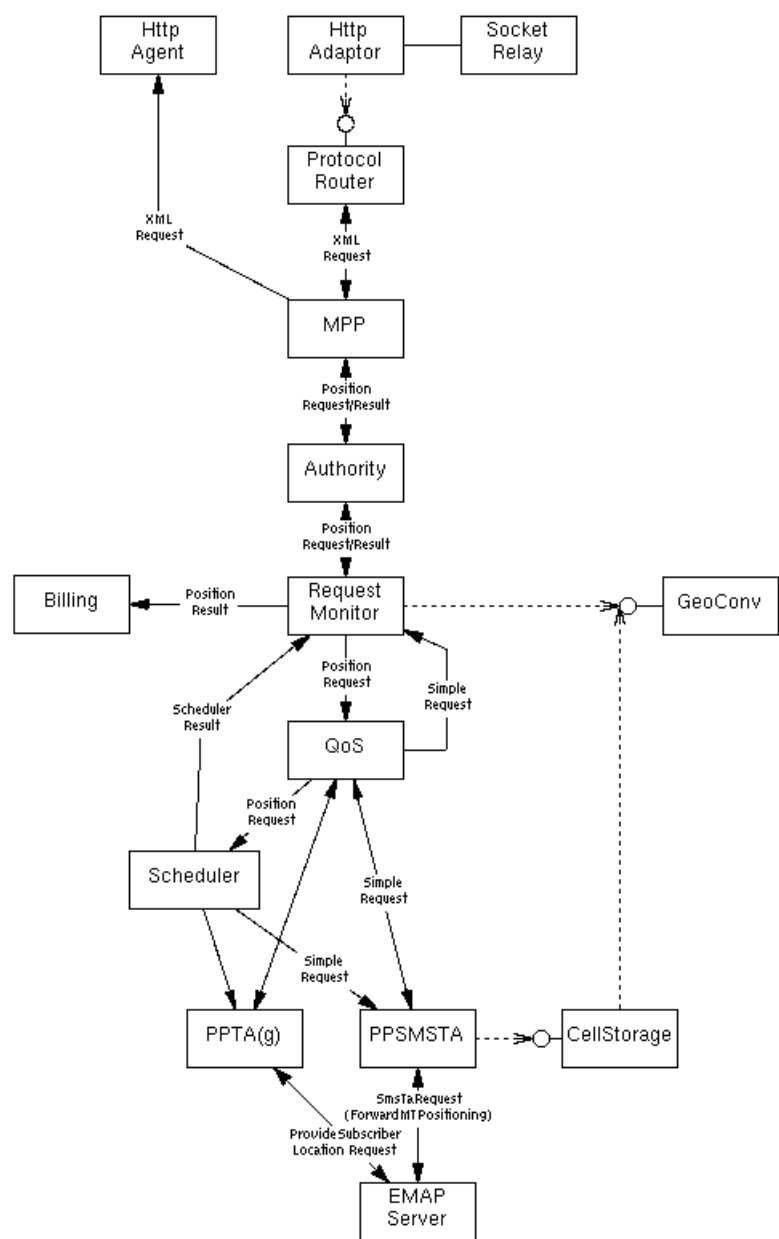
The goal of the analysis of the MPC system was to get a prediction of the costs of modifying the system for changes that were likely in the future. For this we need to find scenarios that are likely to occur in the future of the system, estimates of the amount of change required for each scenario, and a model to derive the prediction. The following sections describe how this works and illustrate the process using the analysis that we performed of the MPC system.

Architecture Description

Before the evaluation of the scenarios may start we need a description of the software architecture. The scenario evaluation method determines the type of information that is required from the description of the software architecture. In case of maintenance prediction we need information to estimate the amount of modifications required to realize the scenario. First, we need to be able to identify in what parts of the architecture a specific function is realized. For this task we may use the logical view (Kruchten 1995), or the conceptual and module view (Hofmeister *et al.* 1999a). After we have determined what functions and parts need to be modified, we determine what other parts may be affected in turn by such changes, i.e. ripple effects. For this task we also need information about the relationships between the parts, which can, for the most part, be found in the aforementioned views. If other views are available that describe aspects, we should consider using them as well.

In addition to the basic information about the architecture we also need size estimates of the components to derive our prediction from. Concerning the choice of the size metric, we should choose the same size metric as already in use for time and effort estimates in the project planning. The size estimates need to conform to the same scale unit as is used for the productivity measures in the project or company, e.g. lines of code, function points (Albrecht, 1979; Albrecht and Gaffney, 1983).

Figure 31.
*The component
view for a part of
the MPC*



In the MPC case, the architecture level design documentation was available for all the basic architecture information described in “Architecture description” on page 58. The architecture descriptions contained the following views:

- System environment that presents the system’s relation to the environment

- Message sequence diagrams that present the behavior for key functions in the system
- An architecture overview presenting the how the system is conceptually organized in layers.
- Component view (Figure 31) that presents the components in the system as they comply with the proprietary component standard.

The size estimates of the MPC components needed for the prediction were obtained by asking the architect to estimate the component sizes for the proposed architecture. The architect used the size data from the previous project as a frame of reference. The estimates were expressed as lines of code in the final implementation of this version of the product. For example, the Http Adaptor component was estimated at 2 KLOC and the Protocol Router at 3 KLOC.

Change Scenario Elicitation

The next step is to elicit a set of change scenarios. When the goal of the analysis is maintenance prediction the aim is to find the scenarios that are most likely to occur in the predicted period. First, we need to decide on the period the prediction should concern, since this sets the scope for the scenario elicitation. The predicted period is often the time from the coming release up to the subsequent release, but may just as well be longer. The explicit prediction period helps the stakeholders in deciding if a scenario may occur during that period as opposed to some unspecified duration in the future where everything could be possible.

We also have to decide which stakeholders to interview for scenarios. When selecting the stakeholders to interview it is important to select persons that have different responsibilities to cover as many relevant angles on the system's future changes as possible. The following stakeholder areas or their equivalents should at least be considered: architecture design, product management, marketing and sales, help desk or customer training, installation and maintenance, and development and implementation.

Projects that develop software for a market where each product may have several customers, e.g. consumer products, seldom

have stakeholder representatives from the customers directly available. In these cases the project is dependent on other knowledge sources, e.g. usability studies, marketing, sales, or support.

If the prediction is part of the design cycle, we elicit the scenarios only for the first iteration and use the same scenario profile for the subsequent predictions. The profile is then updated if additional scenarios are discovered under way.

For the MPC analysis we selected and interviewed the following stakeholders of the MPC to elicit change scenarios:

- the software architect,
- the operative product manager,
- a designer involved in the MPC system's development, and
- an applications designer developing services using the MPC.

When the appropriate stakeholders have been selected we interview the stakeholders. The preferred elicitation technique in this case is bottom-up, since it is assumed that the stakeholders have the better knowledge about likely scenarios. It means that we interview the stakeholders without any particular guidance with respect to what scenarios to find and the interview is concluded when the stakeholders can think of no other likely scenario for the prediction period.

The following scenarios are examples from the total of 32 change scenarios that were elicited during the interviews with the MPC stakeholders:

- Change MPC to provide a digital map with position marked
- Add geodetic conversion methods to the MPC

When the stakeholders have been interviewed we synthesize all the scenarios into one list and arrange them into categories. Duplicate scenarios should be removed. The categories should be specific to the case and help determine if any important categories of scenarios were missed. When we have synthesized and categorized the scenarios from all the stakeholders, we present the complete and categorized list of scenarios to the stakeholders, and ask them to revise the list if they find it

necessary. The revision process is repeated until no stakeholder makes any changes to the set of scenarios or the categorization.

After incorporating the comments from the MPC stakeholders we came to the following five categories of change scenarios:

- Distribution and deployment changes
- Positioning method changes
- Changes to the mobile network
- Changes of position request interface/protocol
- In-operation quality requirements changes

The stakeholders reviewed the revised set of scenarios and the categories with no further comments and thus we reached the stopping criterion.

For the purpose of maintenance prediction, there is one more step to perform. We also need some sort of weight to indicate the probability of the scenarios to occur. We call this the *scenario weight* since it is used in the prediction to determine the scenario's influence on the end result. We either ask one stakeholder, who is the most appropriate one, to estimate the weights, or use a wide band delphi approach (Boehm, 1981) to combine multiple stakeholders' views on the weights.

In both cases we collect the weights for each scenario by asking each stakeholder to estimate the number of times that he or she expects the type of changes represented by the scenario to occur during the prediction period. If we decide on the wide band Delphi approach, we apply that to this step. When the weights have been decided we normalize them, $NW(S_n)$, by dividing the estimated number of changes, or weight, of the scenario, $W(S_n)$, by the sum of the weights of all the scenarios (Equation 7). The result of the normalization should be that all scenarios now have a weight between zero and one and that the sum of all scenario weights is exactly one. We call the list of scenarios with normalized weights the scenario profile.

Equation 7.

Normalizing scenario weights

$$NW(S_n) = \frac{W(S_n)}{\sum_n W(S_N)}$$

To obtain the weights for the MPC scenarios we decided to ask the operative product manager since the future development of the product mainly is the responsibility of that role. The probability estimates were obtained by interviewing the operative product manager and for each scenario in the final scenario profile asking how many times an instance of the scenario was likely to occur. Table 14 contains the weights of two of the scenarios.

Change Scenario Evaluation

After we have elicited a set of change scenarios and their weights, we evaluate their effect. The first step in this evaluation is to determine the effect, or impact, of each scenario in the scenario profile. First, we need to understand the scenario and the functions that are required to realize the scenario. Some of these functions may already be present in the architecture and we need to determine if they must be modified for this scenario, or not. Some functions may need to be added and we should determine where those functions should be added in the architecture; either they can be added to existing components, in which case it is a kind of modification, or they can be added as separate components. To this end, we should consult the software architecture descriptions and possibly the architect if the descriptions are inconclusive. Once the functions have been identified we must trace the effects of the identified changes, because other functions may depend on the modified parts and thus ripple effects may occur.

To use the analysis results in the prediction we must express the impact estimates for each scenario as:

- The size of the modification to existing components. This can be derived from an estimate of the ratio of change for the modified components, e.g. half of the component needs to be changed, and then using the component size to derive the change volume.
- The estimated size of components that need to be introduced.

Obviously, the change volume must be expressed by the same unit of scale as you used in the previous steps, e.g. lines of code, function points or object points.

In the analysis of the MPC, we carefully studied the architecture descriptions and preliminary designs of the components that should be added for a scenario and then estimated:

- The ratio of the estimated modification of each component affected by the scenario.
- The size of any new component added to the architecture.

The results were, per change scenario, a set of estimates of the modification volume for each affected component, as depicted in table 14. Note that the table excludes 30 scenarios for business sensitivity reasons, which makes the figures somewhat different than in the real case.

Table 14: The impact and the calculation of average effort per scenario

ID	Change Scenario	Weight	Impact	
			Modified	New
1	Change MPC to provide a digital map with position marked.	0.0345	25% of 10 kLOC + 10% of 25 kLOC = 5 kLOC	5 kLOC = 5 kLOC
2	New geodetic conversions methods per installation.	0.0023	10% of 10 kLOC + 75% of 5 kLOC = 4.75 kLOC	None
...
Sum:			0.0345 * 5 kLOC + 0.0023 * 4.75 kLOC = ~183 LOC/scenario	0.0345 * 5 kLOC = ~173 LOC/ scenario

We separate the modifications made to existing components from the new components based on the hypothesis that the productivity of writing new code is higher than the productivity of making modifications to existing code.

Interpretation

Prediction of maintenance effort requires a model that is based on the cost drivers of your maintenance processes. Organizations have different strategies for doing maintenance and this affects how the cost and effort relate to the change

traffic and modified components. For instance, Stark and Oman (1997) studied three maintenance strategies and found that there are significant differences in cost between these strategies. To come to the prediction we need to decide on a prediction model. We propose a simple model (Equation 8) that assumes that the change volume is the main cost driver and that we have a productivity figure for the cost of adding new code and modifying old code. For each scenario we have two impact size estimates, changed code, CC, and new code, NC, and a weight. We also have the productivity estimates for these two types of changes. The individual sums of the product of the impact size estimates and the weight of the scenario multiplied with their associated productivity, P_{cc} and P_{nc} , make up the total effort needed for the changes in the scenario profile. To get the average, this number is divided by the number of scenarios in the scenario profile, $C(S)$. The total effort for the period is obtained by multiplying the average effort with the expected number of changes.

Equation 8.

Total maintenance effort calculation

$$E_M = \frac{\sum_{IA} \left(\left(\sum_{CC_j} \text{size}_j \cdot \text{weight}_j \right) \cdot P_{cc} + \left(\sum_{NC_j} \text{size}_j \cdot \text{weight}_j \right) \cdot P_{nc} \right)}{C(MS)} \cdot CT_{\text{estimated}}$$

In the analysis of the MPC system, each release of the system is carried out as its own development project on a regular basis. The strategy is to make changes by designing the new version of the system based on the current one, making the modifications to the code baseline, i.e. the code of the current release, and then testing the system. For this assessment we assume that the overall productivity per software engineer, i.e. not only writing the source statements, in this process is on the same level as those published in the software engineering literature:

- 40 LOC/month for modifying existing code (Henry and Cain 1997), and
- 250 LOC/month for adding new code (Maxwell *et al.* 1996).

These productivity measurements vary between projects and companies and the numbers used here should not be taken as universally applicable. Instead, the productivity data from the

organization that is responsible for the system's maintenance should be used when using this method.

If there are other significant cost drivers in maintenance processes, the prediction model should be modified. When we decide on a different model we should make sure that all the necessary information is available from the previous steps.

Conclusions

In this chapter we have presented the techniques for the steps in ALMA to perform maintenance predictions. Concerning the description, the basic software architecture description is augmented with size estimates. Elicitation is focused on an explicit prediction period and the likely scenarios acquired from the selected stakeholders. Evaluation concerns determining the change volumes by two steps, affected functions and ripple effects. Concerning the prediction we present a prediction model based on the estimated change volume and productivity ratios.

CHAPTER 9: **Fraud Control Centre Case**

This chapter presents the case study in which we applied the architecture level modifiability analysis method (ALMA) to predict the modification effort for the coming release of the Fraud Control Centre product at Ericsson Software Technology AB. The goal with the case study was to apply the techniques for prediction the modification effort as well as the best and worst case. The challenge in this case study was the same as in the mobile positioning case in chapter 8, i.e. that we could not involve all the relevant stakeholders in the process.

The next sub-section briefly describes the mobile telecom fraud control domain. The remainder of the chapter is organized after the steps of the analysis method, starting with a section on the goal setting, followed by sections describing the software architecture, the scenario elicatio, the scenario evaluation and the results interpretation. The last section in this chapter presents the conclusions.

Fraud Control Systems

Usually, cellular telecom operators introduce cellular telephony into an area with a primary concern with establishing network capacity, geographic coverage and signing up customers. However, as subscriber and network growth level cost margins and other financial issues become more important, e.g. lost revenues due to fraud.

Fraud in cellular telecom can be divided into two main categories; subscriber fraud and cloning fraud. Subscriber fraud is when subscribers use the services but does not pay their bills. The two main means to prevent this is to perform subscriber background and credit history checks. Cloning fraud means that a caller uses a false or stolen subscriber identification in order to make calls without paying, or to be anonymous. There exist various approaches to identify and stop cloning. The FCC

system is a part of an anti-fraud system that counter-checks cloning fraud using real-time analysis of network traffic.

The FCC system is a commercial system and because of this we cannot disclose the actual requirements posed on it. For the purpose of illustrating the method we will denote the requirements as R1, R2, etc. We are aware that this is a limitation of the illustration, but argue that the problem is inherent to using a contemporary commercial system. The alternative would be to use an artificial example, or a retired system, but it too has its limitations. In addition, because of size restrictions we are not able to present all details in this illustration, e.g. the complete set of scenarios, or an elaborate description of the impact analysis itself.

System Overview

Software in the switching network centres provides real-time surveillance of suspicious activities associated with a call. The idea is to identify potential fraud calls and have them terminated. However, one single indication is not enough for call termination.

The FCC system allows the cellular operator to specify certain criteria for terminating a suspicious call. The criteria that can be defined are the number of indications that has to be detected within a certain period of time before any action is taken. It is possible to define special handling of certain subscribers and indication types. For the rest of this paper we will use the term 'event' to denote a fraud indication from the switching network.

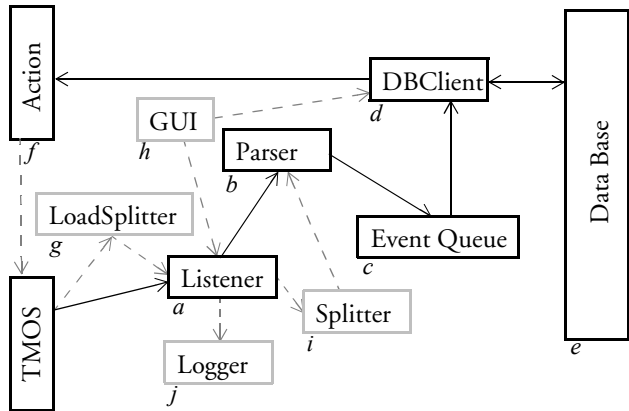
The events are continuously stored in files in the cellular network. With certain time intervals or when the files contain a certain number of events these files are sent to the FCC. The FCC system stores the events and matches them against pre-defined rules. When an event match a rule, the FCC system sends a message to terminate to the switching network and the call is terminated. It is possible to define alternative actions, e.g. to send a notification to an operator console.

Goal Setting

The goal for the evaluation was to predict the modifiability effort as well as finding out the best and worst-case.

The FCC application consists of seven software components, all executing on the same computer (see figure 32). The majority of the components was designed using object oriented techniques and implemented in C++. In one component a scripting language was used in addition to C++. The components do not follow a predefined component model but are rather collections of related classes, a view of components that is common in industry (Bosch, 1999). TMOS is an Ericsson propriety off the shelf component that handles the interaction with the switching network. In this application it is used for collecting event files from switches and other utility services. The *Listener* (*a*) collect events from switches via the TMOS component. The TMOS component is a third party component and is considered only as is, i.e. we cannot make changes to this component. The events are then propagated to one of the executing *Parser* (*b*) threads. The parser extracts the event data from the event files and store then in the *EventQueue* (*c*). Using predefined rules the *DBClient* (*d*) checks the events in the Event Queue and stores the event data in the database. If an event triggers a rule, the *Action* (*f*) component notified, who in turn executes the actions associated with the rule, e.g. a call termination. Standard Unix scripts are used to send terminating messages to the switching network. The components marked *g-j* are components that were judged as potential new components during the assessment, and have been added to this figure to illustrate how the fit in the rest of the architecture.

Figure 32.
The Main software architecture



Change Scenario Elicitation

For the purpose of this study we asked a person related to the FCC project to prepare a list of the most likely changes to the FCC for the future. No other restrictions or instructions were given. We present a selection of the thirteen scenarios from the list we received and used in the assessment below.

S1 New version of the switch software. Could lead to new event types and changed attributes of events.

S2 Port to Windows NT.

S5 Change of data base supplier.

S6 Port of graphical user interfaces to Java.

S8 Interactive termination. An operator must acknowledge call termination.

Change Scenario Evaluation

The scenario elicitation produced thirteen change scenarios that were evaluated using the approach described in “Modifiability Prediction Model” on page 65. Table 15 presents the results from the evaluation. The leftmost column is the id of the change scenarios. Each of the existing components of the architecture is under the heading ‘Components’ and range from A to F. Below the component identifier, is the size estimate for each component. In this case component A’s size has been estimated to 1 kLOC. Under each component is two columns, of which one is shaded. The gray shade column holds the ratio of changed code per scenario and component. The non-shaded column holds the ratio of added lines of code. This can amount to a number higher than 1.

In the evaluation of the scenarios added components are identified as part of the solution, those are listed under ‘New Components’ and range from G-H. They too, have a estimated size. Under New components is only one column per component and it is either nothing or one, i.e. either you add a whole component or none at all.

Table 15: The Assessment Results of the Architecture

SCENARIO	COMPONENT						NEW COMPONENTS				AFFECTED REQUIREMENTS
	A	B	C	D	E	F	G	H	I	J	
	1 kLOC	4 kLOC	1 kLOC	3 kLOC	5 kLOC	1 kLOC	2 kLOC	5 kLOC	2 kLOC	.5 kLOC	
1		.2 .5		.1 .1	.2						r2, r10
2	.3	.1		.2							r3
3	.5 .5		.8 .5								
4				.2	.3		1				r4
5				.4	.5						
6	.5 .5			.3 .3				1			r1,r11,r12
7				.1 .1		.1 .8					r5,r13,r14,r15
8	.2 .2			.4 .4		.1 .5		1			r1, r5, r16
9		.3 .4		.5 .2	.3 .3						r2, r17, r18
10	.3								1		r2
11						.2 .2					r5
12	.2									1	r8
13				.3 .5	.2 .3						r9

Interpretation

The original architecture (figure 32) was assessed with the assessment method described in “Modifiability Prediction Model” on page 65. The components are marked in the figure 32. The results from the scenario impact analyses are presented in the table below (Table 15). For this illustration we choose to set productivity for changing (P_{cc}) to 40 LOC/Man-month (Henry and Cain, 1997), and based on our assumption that writing plug-in code is somewhat easier to do than modifying existing code, we choose to set P_p at 60 LOC/Man-Month. Based on the productivity for writing new code we choose to set P_{nc} at 250 LOC/Man-Month (Maxwell *et al.*, 1996).

The maintainability effort on average per scenario of this architecture according to the calculation in Equation 1 on page 67 and based on the changes scenarios defined on page 164 is 63 man-months. The optimal maintenance effort according to the first equation (Equation 2 on page 69) is 16 Man-Months on average per scenario. The refined equation renders these results instead (Equation 3 on page 70) is 27 Man-Months on average per scenario.

Concluding, the technique we present in this paper allows the software architect to compare the assessed value (63 man-months per change scenario) with the optimal value (27 man-months per change scenario). Based on the difference between these figures, the architect may decide whether this is acceptable or that additional effort has to be spent on improving on the software architecture for maintainability. Since software architecture design requires trade-offs between different quality attributes, the presented technique provides the software architect with more and better information in order to take well-founded decisions.

The work presented in this chapter is motivated by the increasing realization in the software engineering community of the importance of software architecture for fulfilling quality requirements. Traditionally, the software architecture design process is rather implicit and experience driven without a clear and explicit method. One main reason for this is the lack of architecture assessment techniques that quantify quality attributes of the software architecture.

We have illustrated and exemplified the technique by using an industrial case from the telecom domain and found that, according to the method proposed in this paper, the optimal average effort per scenario is about half of the same average effort per scenario of the current architecture. In addition, the most optimistic calculation (Equation 2) suggested only a quarter of the average effort compared to the current architecture. This, however, seem unrealistically optimistic.

CHAPTER 10: Fraktarna Case

This chapter presents the case study in which we applied the architecture level modifiability analysis method (ALMA) to assess the risk associated with future modifications to the EASY system developed by Cap Gemini Sweden for DFDS Fraktarna AB, a Swedish distributor of freight. The goal was to uncover unanticipated risks concerning the modifiability of the EASY system during its expected life-time. The challenge in this case study was to get enough time to interview and discuss with the stakeholders. The reason for this was that the system owner and the developing company were located ~200 kilometers away.

The next sub-section briefly describes the freight handling and tracking domain. The remainder of the chapter is organized after the steps of the analysis method, starting with a section on the goal setting, followed by sections describing the software architecture, the scenario elicitation, the scenario evaluation and the results interpretation. The last section in this chapter presents the conclusions.

Freight Tracking and Handling

The system that we investigated is a business information system called EASY. EASY tracks the handling and location of groupage (freight with a weight between 31 kg and 1 ton) through the company's distribution system. EASY works by uniquely identifying each groupage and the tracking is achieved using bar-code scanners located in different steps of the handling process. The general process is often like the following:

- 1 A customer orders the transport by filling in a form electronically using electronic document interchange (EDI) and prints a unique label that is attached to the groupage.
- 2 A pick up truck arrives at the customer site and picks up the groupage.

- 3 The groupage arrives at the closest terminal and the label is scanned by a bar-code scanner.
- 4 The groupage is then reloaded onto a truck destined for the a hub-terminal. When loaded the bar-code is scanned once again.
- 5 Arriving at the hub-terminal the package is scanned and later when loaded onto another truck destined for the end-terminal, the groupage is scanned once more.
- 6 Arriving at the end-terminal the groupage is scanned and later when loaded on the delivery truck it is scanned one final time.
- 7 Delivery at destination.

The easy system tracks and reports the position of the groupage using scanning points, such as those mentioned in these steps.

Goal Setting

The goal pursued in this case study is risk assessment: we investigate the architecture to identify modifiability risks. This meant that the techniques that we use in the various steps of the analysis are aimed at finding change scenarios that are potentially troublesome to implement.

Architecture Description

To come to a description of EASY's software architecture, we studied the available documentation and interviewed one of the architects and a designer. For risk assessment it is important that the architecture description contains sufficient information to determine whether a change scenario may be troublesome to implement.

The environment should be included in the description because it allows the analyst to take dependencies to external systems into account in the analysis. Some change scenarios require to be implemented that related systems in the environment is adapted as well as the system itself. In such cases it may be a complicating factor that the system's owner has very limited control over the related systems. This may be because

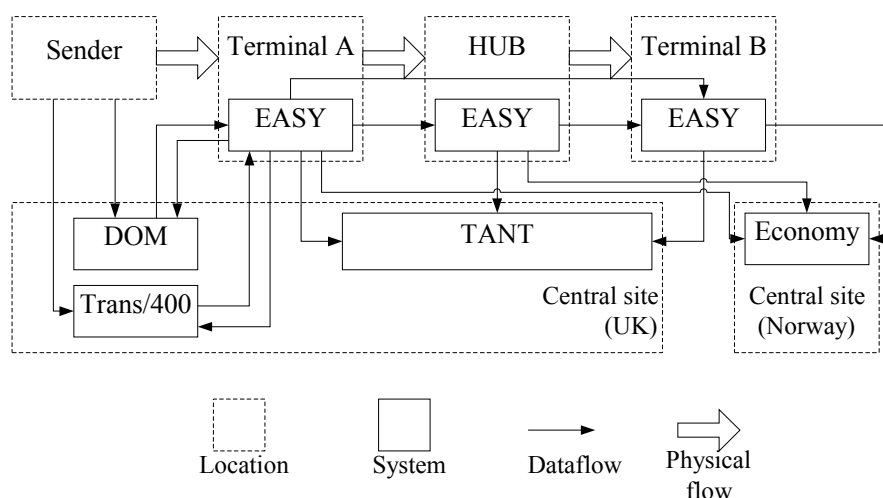
the related systems are owned by other parties, or are legacy systems that are simply too expensive to modify.

Figure 33 shows that EASY communicates with four other systems:

- domestic freight system (DOM) that handles registration of domestic freight
- Trans/400 for registration of international freight
- TANT for storing all location information of groupage
- Economy: system for handling financial data

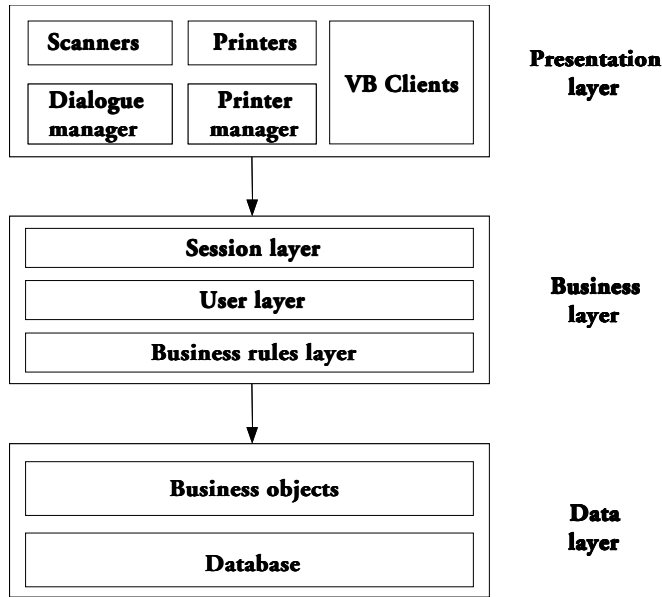
The details of the communication between EASY and these systems should be added in a textual description. For instance, we should add that the communication between the systems takes place through messages using an asynchronous, message-oriented, middleware.

Figure 33.
*EASY in the context
of related systems.*



The conceptual view of EASY is shown in figure 34. This figure shows that the system is divided into three abstract layers: the presentation layer, the business layer and the data layer. The presentation layer consists of all devices that interact with users, the business layer contains all business specific functionality and the data layer manages the persistent data of the system.

Figure 34.
*Conceptual view of
 EASY*



Change Scenario Elicitation

In change scenario elicitation for risk assessment, the aim is to elicit scenarios that expose risks. The first technique that we can use is to explicitly ask the stakeholders to bring forward change scenarios that they think will be troublesome to realize.

Another elicitation technique is that the analyst guides the interviews based on his knowledge of the architectural solution, as obtained in the previous step, and knowledge of usually troublesome changes. Although the interviewer guides the elicitation process, it is important that the stakeholders being interviewed bring the change scenarios forward.

Assisted by this knowledge, the analyst may ask the stakeholders during scenario elicitation for such change scenarios, i.e. ‘Could you think of a change to the system that affects other systems as well?’. Lassing *et al.* (1999c) structured their experiences with troublesome scenarios, as a classification scheme. The classification scheme includes the categories of change scenarios that they found in the domain of business information systems. This scheme assists the analyst and can be used to guide scenario elicitation. The knowledge upon which it is based is most likely domain specific; changes that are

considered complex in one domain are not necessarily troublesome in another domain as well.

These change scenarios may have different stimuli; changes in the functional specification, changes in the requirements, changes in the technical environment, and other sources.

Combining the complex scenarios with the sources from which they originate results in framework of 4 by 5 cells. In the interviews, the analyst tries to cover all the cells of this scheme. For some systems a number of cells in the scheme remain empty, but they should at least be considered. For instance, for a stand-alone system, there will be no scenarios in which the environment plays a role.

For change scenario elicitation in the analysis of EASY we interviewed three stakeholders, i.e. a software architect, a software designer and a representative of the owner of the system. These interviews took approximately one to two hours. In these interviews we employed a top-down approach; we directed the interviews towards complex scenarios using the above-mentioned framework.

One of the change scenarios found in the elicitation step of the analysis of EASY is that the middleware used by the TANT system changes. The effect of this scenario is that EASY has to be adapted to this new middleware as well, because otherwise it will no longer be able to communicate with the TANT system. This change scenario falls in the category 'Change scenarios that are initiated by others than the owner of the system under analysis, but do require adaptations to that system' and its stimuli is a change in the technical environment. This scenario represents the equivalence class of all changes to the middleware, i.e. it is not specific for a special type of middleware. Similarly, we tried to discover change scenarios for all other categories. This process resulted in 21 change scenarios, a number of which are shown in table 16.

Table 16: Examples of change scenarios for EASY

Source	Change scenario
Technical infrastructure	Change of middleware used for TANT
Technical infrastructure	Use of different type of scanners
Technical infrastructure	Change operating system used for EASY
Functional specification	Introduction of new terminals
Requirements	Substantially more parcels to be handled
Requirements	Reducing the number of scan points

The elicitation framework also serves as the stopping criterion for scenario elicitation: we stop the elicitation process when all cells have been considered explicitly. To do so we use an iterative process; after each interview we use the framework to classify the scenarios found. In the next interviews we focus on the cells that are scarcely populated or empty. By going through all cells in the scheme we are able to judge whether all risks are considered.

We used this approach in the analysis of EASY; the end result is shown in Table 17. The table includes a '+' in the cells for which we found one or more change scenarios and a '-' in the cells for which we did not find any change scenario. One of the striking things about this table is that it includes a large number of minus signs. One of the reasons for this is that the integration between EASY and other systems is not very tight. As a result, we found relatively few scenarios associated to the environment.

Change Scenario Evaluation

For risk assessment it is important that the results of the scenarios are expressed in such a way that it allows the analyst to see whether they pose any risks. In (Lassing *et al.*, 1999b) an evaluation model for risk assessment for business information systems is proposed. In that model, the results of scenario evaluation are expressed as a set of measures for both the internal and the external architecture level. The complexity of a change scenario is determined in the following steps:

Table 17: Classification for eliciting complex change scenarios

	Change in the functional specification	Change in the requirements	Change in the technical environment	Other sources
Change scenarios that require adaptations to the system and these adaptations have external effects	+	–	–	–
Change scenarios that require adaptations to the environment of the system and these adaptations affect the system	+	–	+	–
Change scenarios that require adaptations to the external architecture	–	–	–	–
Change scenarios that require adaptations to the internal architecture	+	+	–	–
Change scenarios that introduce version conflicts	–	–	–	–

Initiator of the changes

The first step in the evaluation of the change scenarios is to determine the initiator of the scenario. The initiator is the person or unit in the organization that is responsible for the adaptations required for the scenario.

Impact level

The second step is to determine the extent of the required adaptations. In maintenance prediction, the number of lines of code is used to express the impact, but in risk assessment we limit ourselves to four levels of impact: (1) the change scenario has no impact; (2) the change scenario affects a single component; (3) the change scenario affects several components, and (4) the change scenario affects the software architecture. In expressing the impact, we make a distinction between the effect at the internal architecture level and the effect at the external architecture level, i.e. the effect on the internals and the effect on the environment. At both levels, the same measure is used; at the external architecture level systems are components. The result of this step consists of two measures: the internal

architecture level impact and the external architecture level impact, both expressed using the four-level scale define above.

Multiple owners The next step in the evaluation of the scenario is to determine who is involved in the implementation of the changes required for the scenario. The involvement of multiple owners is indicated using a Boolean value. Components might be owned by other parties and used in other systems and contexts as well. Changes to these components have to be negotiated and synchronized with other system owners.

Version conflicts The final step in the evaluation of the change scenario is to determine whether the scenario leads to different versions of a component and whether this introduces additional complexity. The occurrence of a version conflict is expressed on an ordinal scale: (1) the change scenario introduces no different versions of components, (2) the change scenario does introduce different versions of components or (3) the change scenario creates version conflicts between components. The introduction of different versions is a complicating factor, because it complicates configuration management and requires maintenance of the different versions. Moreover, when the different versions of the components conflict with each other, the scenario might be impossible to implement.

As mentioned, this model originated from work in the area of business information system. In other areas, a different set of measures may be more suitable to expose potential risks.

In the analysis of EASY, we used this model to express the effect of the change scenarios acquired in the previous step. One of the scenarios that we found is that the owner of TANT changes the middleware used in the TANT system. The first step is to determine the initiator of this change scenario. The initiator of this scenario is the owner of TANT.

The next step is to determine the required changes, both at the external architecture level and at the internal architecture level. At the external architecture level, TANT itself and all systems that communicate with it through middleware have to be adapted. Figure 33 shows that EASY communicates with TANT, so it has to be adapted. EASY and TANT have different owners meaning that coordination between these owners is required to implement the changes: a new release of TANT can only be brought into operation when the new release of EASY is finished, and vice versa.

The next step is to investigate the internal architecture of EASY, because the system has to be adapted for this scenario. From the information that we gathered in the first step, it is not apparent which components have to be adapted to implement these changes. We have to consult the architect to find out which components access TANT using TANT’s middleware. It turns out that access to TANT is not contained in one component, which means that several components have to be adapted for this scenario. These components have the same owner, so there is just a single owner involved in the changes at the internal architecture level. The last step is to find out whether the scenario leads to version conflicts. We found that of each component only one version will exist, so there will not be any version conflicts.

We applied the same procedure to the other change scenarios. In table 18 the results of some of them are expressed using the measurement instrument.

Table 18: Evaluation of scenario

Scenario	Initiator	External architecture level			Internal architecture level		
		Impact level	Multiple owners	Version conflicts	Impact level	Multiple owners	Version conflicts
TANT middle-ware replaced	Owner of TANT	3	+	1	3	–	1
Operating system replaced	Owner of EASY	1	–	1	4	–	1
Reduction of scan-points	Owner of EASY	2	+	1	3	–	1

Interpretation

For risk assessment, the analyst will have to determine which change scenarios are risks with respect to the modifiability of the system. It is important that this interpretation is done in consultation with stakeholders. Together with them, the analyst

estimates the likelihood of each scenario and whether the required changes are too complicated. The criteria that are used in this process should be based on managerial decisions by the owner of the system.

We focus on the change scenarios that fall into one of the risk categories identified in the previous section. In the analysis of EASY, for instance, one of the change scenarios that may pose a risk is that the middleware of TANT is changed. This scenario requires changes to systems of different owners including EASY, for which a number of components have to be adapted. Based on that, the stakeholders classified this change scenario as complex. However, when asked for the likelihood of this scenario they indicated that the probability of the change scenario is very low. As a result, the change scenario is not classified as a risk. Similar was done for the other scenarios and the conclusion was that two of the scenarios found could be classified as risks. For these risks various risk mitigation strategies are possible: avoidance (take measures to avoid that the scenario will occur or take action to limit their effect, for instance, by use of code-generation tools), transfer (e.g. choose another software architecture) and acceptance (accept the risks).

Conclusions

This section concerns risk assessment using architecture analysis. We have presented techniques that can be used in the various steps of the analysis and illustrated these based on a case study of modifiability analysis we performed. For change scenario elicitation we have presented a classification framework, which consists of categories of change scenarios that have complicated associated changes. The framework is closely tied to the scenario evaluation instrument that we discussed.

The viewpoints, the classification framework and the evaluation instrument result from experiences with architecture analysis (Lassing *et al.*, 1999b, 1999c, 2000). However, for interpretation we do know which information is required but we do not have a frame of reference to interpret these results, yet. In its current form this step of the method relies on the stakeholders and the analyst's experience to determine whether a change scenario poses a risk or not.

CHAPTER 11: Software Architecture Analysis Experiences

This chapter describes the experiences acquired during the definition and use of ALMA. The presentation of these experiences organised around the five steps of the method and a section for more general experiences. Examples from the two previously presented case studies, (chapter 8 and chapter 10) illustrate the experiences.

Experience concerning the analysis goal

Modifiability analysis requires one clear goal

At the time of planning these case studies in our first respective meetings with the companies, we discussed the goal of the software architecture analysis. At that time it was not an explicitly stated item on the agenda, but nevertheless was it an implicit part of the discussion. We, the analysts, did not fully understand the impact of the analysis goal to the following steps at that time. But, during the course of the studies we found how much it affected the choice of techniques in the later steps.

The techniques in the following steps of the analysis method are different in significant ways for important reasons. We found that it is tempting to go for more than one analysis goal in 'while you are at it'-kind of reasoning. It is, however, very hard to combine the techniques, or, to perform the steps of the analysis using, for example, elicitation techniques for different goals in parallel.

The goal of the analysis determines the techniques to be used in the following analysis steps. Be sure to have only *one* clear goal set at the very beginning of the analysis.

Set of Views to use

We found that in software architecture analysis of modifiability a number of these views is required. The goal of the architecture description is to provide input for the following steps of the analysis or, more specifically, for determining the changes required for implementing the change scenarios. We found that the views most useful for doing so are the views that shows the architectural approach taken for the system, i.e. the conceptual view, and the view that shows the way the system is structured in the development environment, i.e. the development view. However, to explore the full effect of a scenario it is not sufficient to look at just one of these.

For instance, in our analysis of EASY the owner of the system mentioned a scenario that, from a systems management perspective, it is probably too expensive to have an instance of EASY at all terminals. He indicated that the number of instances should be reduced, while maintaining the same functionality. To determine the effect of this scenario, we had to look at the view of the architecture that showed the division of the system in loosely coupled local instances, i.e. the view that showed the architectural approach for the system. The architect indicated that to implement this scenario there should be some kind of centralization, i.e. instead of an autonomous instance of the system at each site we would get a limited number of instances at centralized locations. This meant that one important assumption of the system was no longer valid, namely the fact that an instance of EASY only contains information about groupage that is processed at the freight terminal where the instance is located. To explore the effect of this change, we investigated the development view and found that a number of components had to be adapted to incorporate this change. So, one view was not sufficient to explore the effect of the scenario, but we did not have to use any views describing the dynamics of the system.

However, for some goals we needed additional information. This issue is discussed in subsequent experiences.

Need for additional information

Another observation that we made is that for some goals we need additional information in our analysis that is normally not seen as part of the software architecture. For instance, for maintenance prediction, we want to make estimates of the size of the required changes. To do so, we need not only information about the structure of the system, but also concerning the size of the components. These numbers are normally not available at the software architecture level, so they have to be estimated by the architects and/or designers.

For risk assessment of business information systems we need another type of additional information. In that case, we need to know about the system owners that are involved in a change (Lassing *et al.*, 1999). This information is normally not included in the software architecture designs, so it has to be obtained separately from the stakeholders.

Scenario elicitation Experiences

The architect's bias

We have experienced a particular recurring type of bias when interviewing the architect of the system being assessed. In both case studies, when asked to come up with change scenarios, the architect suggested scenarios that he had already anticipated when designing the architecture.

Of course, this is no surprise. A good architect anticipates for the most probable changes that the system will undergo in the future. Another reason for this phenomenon could be that the architect of the system is, implicitly, trying to convince himself and his environment that he has made all the right decisions. After all, it is his job to devise a modifiable architecture. It requires a special kind of kink to destroy what you yourself have just created. This is the same problem programmers have when testing their own code. It requires skill and perseverance of the analyst to take this mental hurdle and elicit relevant scenarios from this stakeholder.

For example, in the analysis of EASY, the architect mentioned that the number of freight terminals could change. This type of change is very well supported by the chosen architecture solution. Similarly, in the analysis of the MPC, the architect men-

tioned the change scenario ‘physically divide the MPC into a serving and a gateway MPC’, which was in fact already supported by the architecture. So, relying only on the architect to come up with change scenarios may lead us to believe that all future changes are supported.

Different perspectives

Scenario elicitation is usually done by interviewing different stakeholders of the system under analysis (Abowd *et al.* 96). We found that it is important to have a mix of people from the ‘customer side’ and from the ‘development side’. Different stakeholders have different goals, different knowledge, different insights, and different biases. This all adds to the diversity of the set of scenarios.

In our analysis of EASY, we found that the customer attached more importance on scenarios aimed at decreasing the cost of ownership of the system, e.g. introduce thin-clients instead of PCs. The architect and the designer of the system focused more on scenarios that aimed for changes in growth or configuration, e.g. the number of terminals change and integration with new suppliers’ systems. Hence, to get a more complete picture and reduce the risk of getting a biased (see previous section) it is extremely important to collect scenarios from several different stakeholders.

Structured scenario elicitation

When interviewing stakeholders, recurring questions are:

- Does this scenario add anything to the set of scenarios obtained so far?
- Is this scenario relevant?
- In what direction should we look for the next scenario?
- Did we gather enough scenarios?

Unguided scenario elicitation relies on the experience of the analyst and stakeholders in architecture assessment. The elicitation process then stops if there is mutual confidence in the quality and completeness of the set of scenarios. This may be termed the *empirical* approach (Carroll and Rosson, 1992). One of the downsides of this approach that we have experienced is that the stakeholders’ horizon of future changes

is very short. Most change scenarios suggested by the stakeholders relate to issues very close in time, e.g. anticipated changes in the current release.

To address this issue we have found it very helpful to have some organizing principle while eliciting scenarios. This organizing principle takes the form of a, possibly hierarchical, classification of scenarios to draw from. For instance, in our risk assessment of EASY, we used a categorization of high-risk changes as the following (defined a-priori):

- *Internal* changes with *external* effects.
- *External* changes with *external* effects.
- Scenarios that require changes to the *macro* architecture.
- Scenarios that require changes to the *micro* architecture.
- Scenarios that introduce version conflicts.

This categorization has been developed over time, while gaining experience with this type of assessment. When the focus of the assessment is to estimate maintenance cost, the classification tends to follow the logical structure of the domain of the system, as the following (based on the domain):

- Distribution and deployment changes
- Position method changes
- Mobile network changes
- Positioning interface changes
- In-operation quality requirements changes

If these categories are not known a priori, an iterative scheme of gathering scenarios, structuring those scenarios, gathering more scenarios, etc., can be followed. In either case, this approach might be called *analytical*.

In the analytical approach, the classification scheme is used to guide the search for additional scenarios. At the same time, the classification scheme defines a stopping criterion: we stop searching for yet another change scenario if: (1) we have explicitly considered all categories from the classification scheme, and (2) new change scenarios do not affect the classification structure.

Scenario probability confusion

The maintenance prediction depends on estimating the probabilities for each scenario, which we in ALMA call scenario weights. Of course, the sum of these weights, being probabilities, should be one. The problem for the stakeholders is that they often know that a certain change *will* occur. Intuitively, they feel that the weight for that change scenario should be one. Problems then arise if another scenario *will also* occur.

The key to understanding the weighting is to perform it in two steps, estimation and normalization (see table 19). The prediction is aimed for a certain period of time, e.g. between two releases of the system. If there is a scenario, say scenario 1, that we know will happen, we first estimate how many times changes in the scenario's equivalence class will occur. Suppose one change belonging to the equivalence class will occur. For another change scenario, scenario 2, we expect changes of that equivalence class to occur, say, three times during the same period. This estimation is done for all scenarios. Then we perform step two, normalization. The weights are calculated for each scenario by dividing the number changes estimated for that scenario by the sum of the estimates of the complete set of scenarios. In the example just given, the result is that the weight of the first scenario is 0.25 and the weight of the second scenario is 0.75 (rightmost column in table 19).

Table 19: Weighting process

	Step One	Step Two	Result
	Estimated number of changes	Normalization	
Scenario 1	1	1 / (1+3)	1/4
Scenario 2	3	3 / (1+3)	3/4

Scenario evaluation Experiences

Ripple effects

The activity of scenario evaluation is concerned with determining the effect of a scenario on the architecture. For instance, if the effects are small and localized, one may conclude that the architecture is highly modifiable for at least

this scenario. Alternatively, if the effects are distributed over several components, the conclusion is generally that the architecture supports this scenario poorly.

However, how does one determine the effects of a scenario? Often, it is relatively easy to identify one or a few components that are directly affected by the scenario. The affected functionality, as described in the scenario, can directly be traced to the component that implements the main part of that functionality.

The problem that we have experienced in several cases is that *ripple effects* are difficult to identify. Simply put, a ripple effect is when a component directly affected by a change scenario needs to be changed, and those changes require other components to change. Even if the provided and/or required interfaces of the component does not change, the change may affect, for example, the valid range of parameter values which often is not covered by the interface specifications and thus require changes to the components connected to these interfaces. Such ripple effects are extremely hard to foresee, even though the software architect, during software architecture design, has a good understanding of the decomposition of functionality.

The main factor influencing this problem is the amount of detail present in the description of the software architecture. This is determined by the amount of detail available to the software architect. In a study Lindvall and Runesson (1998) have shown that even detailed design descriptions lack information necessary for performing an accurate analysis of ripple effects. Since less detail is available at the software architecture level, this is an even more relevant problem. In addition, Lindvall and Sandahl (1998) showed that even experienced developers clearly underestimate the number of classes impacted by a change. On the other hand, the classes they did identify as needing change were correct in most of the cases.

To illustrate this problem, consider the following example. In the MPC case, one of the scenarios concerned the implementation of support for standardized remote management. Although this change could be evaluated by identifying the components directly affected, it was extremely hard to foresee the changes that would be required to the

interface of these components. As a consequence, there was a very high degree of uncertainty as to the exact amount of change needed.

Experiences with the results interpretation

Lack of frame of reference

Once scenario evaluation has been performed, we have to associate conclusions with these results. The process of formulating these conclusions we refer to as *interpretation*. The experience we have is that generally we lack a frame of reference for interpreting the results of scenario evaluation and often there are no historical data to compare with.

For instance, in maintenance prediction, the result is a prediction of the average size of a change, in lines of code, function points, or some other size measure. In the analysis of the MPC we came to the conclusion that, on average, there would be 270 lines of code affected per change scenario. Is this number 'good' or 'bad'? Can we develop an architecture for MPC that requires an average of 100 LOC per change? To decide if the architecture is acceptable in terms of modifiability, a frame of reference is needed.

Role of the owner in interpretation

Our identification of the above leads us immediately to the next experience: the *owner of the system* for which the software architecture is designed plays a crucial role in the interpretation process. In the end, the owner has to decide whether the results of the assessment are acceptable or not. This is particularly the case for one of the three possible goals of software architecture analysis, i.e. risk assessment. The scenario evaluation will give insight in the boundaries of the software architecture with respect to incorporating new requirements, but the owner has to decide whether these boundaries, and associated risks, are acceptable or not.

Consider, for instance, the change scenario for EASY that describes the replacement of the middleware used for EASY. This scenario not only affects EASY, but also the systems with which EASY communicates. This means that the owners of these systems have to be convinced to adapt their systems. This may not be a problem, but it *could* be. Only the owner of the

system can judge such issues and therefore it is crucial to have him/her involved in the interpretation.

The system owner also plays an important role when other goals for software modifiability analysis are selected, i.e. maintenance cost prediction or software architecture comparison. In this case, the responsibility of the owner is primarily towards the change profile that is used for performing the assessment. The results of the scenario evaluation are accurate to the extent the profile represents the actual evolution path of the software system.

General experiences

Software architecture analysis is an ad hoc activity

Three arguments are used for explicitly defining a software architecture (Bass *et al.*, 1998). First, it provides an artifact that allows for discussion by the stakeholders very early in the design process. Second, it allows for early assessment or analysis of quality attributes. Finally, it supports the communication between software architects and software engineers since it captures the earliest and most important design decisions. In our experience, the second of these is least applied in practice. The software architecture is seen as a valuable intermediate product in the development process, but its potential with respect to quality assessment is not fully exploited.

In our experiences, software architecture analysis is mostly performed on an ad hoc basis. We are called in as an external assessment team, and our analysis is mostly used at the end of the software architecture design phase as a tool for acceptance testing ('toll-gate approach') or during the design phase as an audit instrument to assess whether the project is on course. In either case, the assessment is not solidly embedded in the development process. As a consequence, earlier activities do not prepare for the assessment, and follow-up activities are uncertain.

If software architecture analysis were an integral part of the development process, earlier phases or activities would result in the necessary architectural descriptions, planning would include time and effort of the assessor as well as the stakeholders being interviewed, design iterations because of

findings of the assessment would be part of the process, the results of an architecture assessment would be on the agenda of a project's steering committee, and so on. This type of embedding of software architecture analysis in the development process, which is in many ways similar to the embedding of design reviews in this process, is still very uncommon.

Accuracy of analysis is unclear

A second general experience is that we lack means to decide upon the accuracy of the results of our analysis. We are not sure whether the numbers that maintenance prediction produces (such as the ones mentioned in chapter 9) are accurate, and whether risk assessment gives an overview of all risks. On the other hand, it is doubtful whether this kind of accuracy is at all achievable. The choice for a particular software architecture influences the resulting system and its outward appearance. This in turn will affect the type of change requests put forward. Next, the configuration control board, which assesses the change requests, will partly base its decisions on characteristics of the architecture. Had a different architecture been chosen, then the set of change requests proposed and the set of change requests approved would likely be different. Architectural choices have a feedback impact on change behavior, much like cost estimates have an impact on project behavior (Abdel-Hamid and Madnick, 1986).

A further limitation of this type of architecture assessment is that it focuses on aspects of the product only, and neglects the influence of the process. For instance, Avritzer and Weyuker (1998) found that quite a large number of problems uncovered in architecture reviews had to do with the process, such as not clearly identified stakeholders, unrealistic deployment date, no quality assurance organization in place, and so on.

Conclusions

In this chapter we presented experiences from applying the analysis method in the previously described case studies.

With respect to the first step of the analysis method, goal setting, we found that it is important to decide on *one goal* for the analysis. For the second step of the method, architecture description, we experienced that the impact of a change

scenario may span several architectural views. However, we also found that views relating to the system's dynamics are not required in modifiability analysis. But for some analysis goals we need information that is not included in existing architecture view models. For risk assessment, we found the system's environment and information about system owners useful in evaluating change scenarios, whereas these are not required when performing maintenance prediction. However, for maintenance prediction, we require information about the size of the components, which is normally not considered to be part of the software architecture design.

Concerning the third step of the method, scenario elicitation, we made the following four main observations. First, it is important to interview different stakeholders to capture scenarios from different perspectives. We also found that the architect has a certain bias in proposing scenarios that have already been considered in the design of the architecture. Another observation we made was that the time horizon of stakeholders is rather short when proposing change scenarios and that guided elicitation might help in improving this. A more specific experience was that the weighting scheme used for maintenance prediction appeared somewhat confusing to the stakeholders.

Concerning the fourth step of the method, scenario evaluation, we experienced that ripple effects are hard to identify since they are the result of details not yet known at the software architecture level. Regarding the fifth step of the method, interpretation, we experienced that the lack of a frame of reference makes the interpretation less certain, i.e. we are unable to tell whether the predicted effort is relatively high or low, or whether we captured all or just a few risks. Because of this, the owner plays an important role in the interpretation of the results.

We also made some general experiences that are not directly related to one of ALMA's steps. First, we have found that, in practice, software architecture analysis is an ad hoc activity that is not explicitly planned for. Second, the validity of the analysis is unclear as to the accuracy of the prediction and the completeness of the risk analysis.

Cited References

Abdel-Hamid T.K. and Madnick S.E. 1986. Impact of Schedule Estimation on Software Project Behavior. *IEEE Software*, 3(4), pp. 70-75.

Abowd G., Bass L., Clements P., Kazman R., Northrop L. and Zarnowski A. 1996. *Recommended Best Industrial Practice for Software Architecture Evaluation*, Technical Report (CMU/SEI-96-TR-025), Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Albrecht A.J. 1979. Measuring application development productivity. In GUIDESHARE: Proceedings of the IBM Applications Development Symposium (Monterey, Calif.), pp. 83- 92

Albrecht A.J and Gaffney, J. 1983. Software function, source lines of code, and development effort prediction; A software science validation. *IEEE Trans Softw. Eng.* 9(6), pp. 639- 648.

Alonso A., Garcia-Valls M., and de la Puente J.A. 1998. 'Assessment of Timing Properties of Family Products,' *Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families*, F. vd. Linden (editor), LNCS 1429, pp. 161-169.

Argyris C., Putnam R. and Smith D. 1985. *Action Science: Concepts, methods, and skills for research and intervention*, Jossey-Bass, San Francisco.

Avritzer A. and Weyuker E.J. 1998. Investigating Metrics for Architectural Assessment. *Proceedings of the Fifth International Software Metrics Symposium*, Bethesda, Maryland, pp. 4-10.

Basili V.R., Selby R.W. and Hutchens D.H. 1986. Experimentation in Software Engineering. *IEEE Transactions on Software Engineering*, 12(7), pp. 733-743.

Baskerville R., 1999. Investigating Information Systems with Action Research, in *Communications of AIS*, 2(19).

Bass L., Clements P. and Kazman R. 1998. *Software Architecture in Practice*, Reading, MA: Addison Wesley Longman.

Bengtsson P. and Bosch J. 1998. Scenario-based Software Architecture Reengineering. *Proceedings of the 5th International Conference on Software Reuse*. Los Alamitos, CA: IEEE Computer Society Press, pp. 308-317.

- Bengtsson P. 1998. Towards Maintainability Metrics on Software Architecture: An Adaptation of Object-Oriented Metrics. *Proceedings of First Nordic Workshop on Software Architecture*. Reasearch Report 1998:14 ISSN: 1103-1581, Blekinge Institute of Technology, Ronneby, Sweden, pp. 87-91.
- Bengtsson P. and Bosch J. 1999a. Architecture Level Prediction of Software Maintenance. *Proceedings of 3rd EuroMicro Conference on Maintenance and Reengineering*, Los Alamitos, CA: IEEE Computer Society Press. pp. 139-147.
- Bengtsson P. and Bosch J. 1999b. Haemo Dialysis Software Architecture Design Experiences. *Proceedings of the 21st International Conference on Software Engineering*. Los Angeles, CA: ACM Press, pp. 516-525.
- Bengtsson P. and Bosch J. 2000. An Experiment on Creating Scenario Profiles for Software Change. *Annals of Software Engineering*, Bussum, Netherlands: Baltzer Science Publishers, vol. 9, pp. 59-78.
- Bengtsson P., Lassing N., Bosch J. and van Vliet H. 2000. *Analyzing Software Architectures for Modifiability*. Technical Report (HK-R-RES-00/11-SE), University of Karlskrona/Ronneby, Ronneby. Submitted for publication.
- Boehm B.W. and In H. 1996. Identifying Quality-Requirements Conflicts. *IEEE Software*, 13(2): pp. 25-24.
- Boehm B.W. 1986. A Spiral Model of Software Development and Enhancement. *ACM Software Engineering Notes*. New York, NY:ACM Press. 11(4), pp. 22-42.
- Boehm, B. W. 1981. *Software Engineering Economics*, Englewood Cliffs, New Jersey: P T R Prentice Hall. ch. 22, p. 335.
- Bohner S.A. 1991. Software Change Impact Analysis for Design Evolution. *Proceedings of 8th International Conference on Maintenance and Re-engineering*. Los Alamitos, CA: IEEE Computer Society Press, pp. 292-301.
- Booch G. 1999. *Object-Oriented Analysis and Design with Applications* (2nd edition). Redwood City, CA: Benjamin/Cummings Publishing Company.
- Booch G., Rumbaugh J. and Jacobson I. 1998. *The Unified Modeling Language User Guide*, Object Technology Series, Reading, MA: Addison-Wesley.
- Bosch J. 2000. *Design & Use of Software Architectures. Adopting and evolving a product-line approach*. Harlow, England: Addison-Wesley.
- Bosch J. 1999a. Design of an Object-Oriented Measurement System Framework. *Domain-Specific Application Frameworks*, M. Fayad, D. Schmidt, R. Johnson (eds.), New York, NY: John Wiley, pp. 177-205.
- Bosch J. 1999b. 'Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study.' *Proceedings of the First Working IFIP Conference on Software Architecture*.
- Bosch J. and Bengtsson P. 2001. Assessing Optimal Software Architecture Maintainability. *Proceedings of Fifth European Conference on Software*

Maintenance and Reengineering. Los Alamitos, CA: IEEE Computer Society Press. pp. 168-175.

Bosch J. and Molin P. 1999. Software Architecture Design: Evaluation and Transformation. *Proceedings of IEEE Engineering of Computer Based Systems Symposium*, Los Alamitos, CA: IEEE Computer Society Press. pp. 4-10.

Bosch J., Molin P., Mattsson M., Bengtsson P., and Fayad M. 1999. Object-oriented Frameworks: Problems and Experiences. *Building Application Frameworks*, M. Fayad, D. Schmidt, R. Johnson (eds.), New York, NY: John Wiley. pp. 55-82.

Briand L., Arisholm E., Counsell S., Houdek F. and Thévenod-Foss P. 1999. Empirical studies of Object-Oriented Artifacts, Methods, and Processes: State of The Art and Future Directions', *Empirical Software Engineering*, 4 (4), pp. 387-404.

Briand L., El Emam K., Freimut B., Laitenberger O. 2000. 'A Comprehensive Evaluation of Capture-Recapture Models for Estimating Software Defect Content', *IEEE Transactions on Software Engineering*, IEEE Computer Society Press, Vol. 26, No. 6, pp 518-540.

Buschmann F., Meunier R., Rohnert H., Sommerlad P. and Stahl M. 1996. *Pattern-Oriented Software Architecture - A System of Patterns*. Chichester, England: John Wiley & Sons.

Carrière J., Kazman R. and Woods S. 1999. Assessing and Maintaining Architectural Quality. *Proceedings of The Third European Conference on Software Maintenance and Reengineering*, Los Alamitos, CA: IEEE Computer Society. pp. 22-30.

Carroll J.M. and Rosson M.B. 1992. Getting around the Task-Artifact cycle. *ACM Transactions on Information Systems*, 10 (2), pp. 181-212.

Clements P., Kazman R. and Klein M. 2002. Evaluation Software Architectures: Methods and Case Studies. SEI series in Software Engineering, Boston, MA; Addison-Wesley.

Clements P. 1996. "A Survey of Architecture Description Languages", *Eighth International Workshop on Software Specification and Design*, Germany.

Dekleva S.M. 1992. The Influence of the Information Systems Development Approach on Maintenance. *MIS Quarterly*, 16 (3), pp. 355-372.

D'Ippolito R.S. 1990. *Proceedings of the Workshop on Domain-Specific Software Architectures*. CMU/SEI-88-TR-30, Software Engineering Institute.

Dueñas J.C., de Oliveira W.L. and de la Puente J.A. 1998. A Software Architecture Evaluation Method. *Proceedings of the Second International ESPRIT ARES Workshop*, Lecture Notes in Computer Science 1429, Berlin, Germany: Springer Verlag. pp. 148-157.

Ecklund Jr. E.F., Delcambre L.M.L., and Freiling M.J. 1996. Change Cases: Use Cases that Identify Future Requirements. *Proceedings OOPSLA '96*, New York, NY: ACM Press. pp. 342-358.

- Fenton N.E. and Pfleeger S.L. 1996. *Software Metrics - A Rigorous & Practical Approach* (2nd edition), London, England: International Thomson Computer Press.
- Gamma E., Helm R., Johnson R., and Vlissides J. 1995. *Design Patterns Elements of Reusable Design*. Reading, MA: Addison-Wesley.
- Harrison R. 1987. Maintenance Giant Sleeps Undisturbed in Federal Data Centers. *Computerworld*, March 9.
- Henry J.E., Cain J.P. 1997. A Quantitative Comparison of Perfective and Corrective Software Maintenance. *Journal of Software Maintenance: Research and Practice*, Chichester, England: John Wiley & Sons, 9(5), pp. 281-297.
- Hofmeister C., Nord R.L. and Soni D. 2000. *Applied Software Architecture*, Reading, MA: Addison Wesley Longman.
- Hofmeister C., Nord R.L. and Soni D. 1999. Describing Software Architecture with UML, *Software architecture: Proceedings of the 1st Working IFIP Conference on Software Architecture*, Donohoe P. (ed.). Dordrecht, The Netherlands: Kluwer Academic Publishers, pp. 145-159.
- Häggander D., Bengtsson P., Bosch J. and Lundberg L. 1999. 'Maintainability Myth Causes Performance Problems in Parallel Applications,' *Proceedings of IASTED 3rd International Conference on Software Engineering and Applications*, pp. 288-294.
- IEEE Architecture Working Group. 2000. *Recommended practice for architectural description*. IEEE Standard 1471-2000.
- IEEE. 1990. IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 610.12-1990.
- ISO/IEC. 2000. *Information technology - Software product quality -Part 1: Quality model*, ISO/IEC FDIS 9126-1:2000(E)
- Jacobson I., Christerson M., Jonsson P. and Övergaard G. 1992. *Object-oriented software engineering. A use case driven approach*. Readings, MA: Addison-Wesley.
- Kamsties E. and Lott C.M. 1995. An Empirical Evaluation of Three Defect-Detection Techniques. In: Schäfer, W., Botella, P. (Eds.), *Software Engineering -- ESEC '95*, pp. 362-383.
- Karlsson E. (ed.). 1995. *Software Reuse A Holistic Approach*. Chichester, England: John Wiley & Sons.
- Kazman R., Abowd G., Bass L. and Clements P. 1996. Scenario-Based Analysis of Software Architecture. *IEEE Software*, 13 (6), pp. 47-56.
- Kazman R., Bass L., Abowd G. and Webb M. 1994. SAAM: A Method for Analyzing the Properties of Software Architectures. *Proceedings of the 16th International Conference on Software Engineering*, New York, NY: ACM Press, pp. 81-90.
- Kazman R., Klein M., Barbacci M., Longstaff T., Lipson H. and Carrière S.J. 1998. The Architecture Tradeoff Analysis Method. *Proceedings*

of 4th International Conference on Engineering of Complex Computer Systems, Monterey, CA: IEEE Computer Society Press. pp. 68-78

Kazman R., Barbacci M., Klein M. and Carrière S.J. 1999. Experience with Performing Architecture Tradeoff Analysis. *Proceedings of the 21st International Conference on Software Engineering*. New York, NY: ACM Press. pp. 54-63.

Kazman R., Klein M. and Clements P. 2000a. *ATAM: Method for Architecture Evaluation*. SEI Technical Report, (CMU/SEI-2000-TR-004).

Kazman R., Carrière S.J. and Woods S.G. 2000b. Toward a Discipline of Scenario-based Architectural Engineering. *Annals of Software Engineering*, Bussum, Netherlands: Baltzer Science Publishers, vol. 9, pp. 5-33.

Kiczales G. 1996. Aspect-oriented programming. *ACM Computing Surveys*, New York, USA: ACM Press. 28(4).

Kruchten P.B. 1995. The 4+1 View Model of Architecture. *IEEE Software* 12 (6), pp. 42-50.

Kung D., Gao J., Hsia P., Wen F., Toyoshima Y. and Chen C. 1994. Change Impact in Object Oriented Software Maintenance. *Proceedings of the International Conference on Software Maintenance*. Los Alamitos, CA:IEEE Computer Society Press, pp. 202-211.

Lassing N., Rijsenbrij D. and van Vliet H. 1999a. Flexibility of the ComBAD architecture. In P. Donohoe (ed.) *Software architecture: Proceedings of the 1st Working IFIP Conference on Software Architecture*. Dordrecht, The Netherlands: Kluwer Academic Publishers, pp. 341-355.

Lassing N., Rijsenbrij D. and van Vliet H. 1999b. Towards a broader view on software architecture analysis of flexibility. *Proceedings of the 6th Asia-Pacific Software Engineering Conference*, Los Alamitos, CA:IEEE Computer Society Press. pp. 238-245.

Lassing N., Rijsenbrij D. and van Vliet H. 1999c. The goal of software architecture analysis: confidence building or risk assessment. *Proceedings of the 1st Benelux conference on state-of-the-art of ICT architecture*, Amsterdam, Netherlands: Vrije Universiteit.

Lassing N., Rijsenbrij D. and van Vliet H. 2000. *Scenario Elicitation in Software Architecture Analysis*. Submitted for publication.

Lassing N., Rijsenbrij D. and van Vliet H. 2001. Viewpoints on Modifiability. Accepted for publication in the *International Journal of Software Engineering and Knowledge Engineering*, Singapore, Singapore: World Science Publications.

Lassing N., Bengtsson P., van Vliet H. and Bosch J. 2001. Experiences with ALMA: Architecture-Level Modifiability Analysis. Accepted for publication in *Journal of Systems and Software*. New York, NY: Elsevier Science Inc.

Li W. and Henry S. 1993. OO Metrics that Predict Maintainability. *Journal of Systems and Software*. New York, NY: Elsevier Science Inc. 23(1), pp. 111-122.

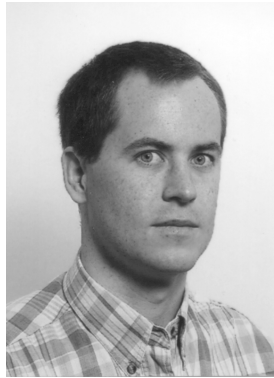
- Lientz B. and Swanson E. 1980. *Software Maintenance Management*. Reading, MA: Addison-Wesley.
- Lindvall M. and Sandahl K. 1998. How Well do Experienced Software Developers Predict Software Change? *Journal of Systems and Software*. 43(1), pp. 19-27.
- Lindvall M. and Runesson M. 1998. The Visibility of Maintenance in Object Models: An Empirical Study. *Proceedings of International Conference on Software Maintenance*. Los Alamitos, CA: IEEE Computer Society Press, pp. 54-62.
- Liu J.W.S. and Ha R. 1995. Efficient Methods of Validating Timing Constraints. *Advanced in Real-Time Systems*, S.H. Son (ed.), Upper Saddle River, NJ:Prentice Hall, pp. 199-223.
- Luckham D.C., Kenney J.J., Augustin L.M., Vera J., Bryan D. and Mann W. 1995. Specification and Analysis of System Architecture Using Rapide, *IEEE Transactions on Software Engineering*. 21(4), pp. 336-355.
- Maranzano J. 1993. *Best Current Practices: Software Architecture Validation*. AT&T Report.
- Maxwell K. D., Van Wassenhove L. and Dutta S. 1996. Software Development Productivity of European Space, Military, and Industrial Applications. *IEEE Transactions on Software Engineering*. 22 (10), pp. 706-718.
- McCall J.A., Quality Factors. 1994. *Software Engineering Encyclopedia*. J.J. Marciniak ed., New York, NY: John Wiley & Sons, Vol 2, pp. 958 - 971
- McCall J.A., Richards P.K. and Walters G.F. 1977. *Factors in Software Quality*. RADC-TR-77-369, US Dept. of Commerce.
- Medvidovic N. and Taylor, R.N. 2000. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*. 26 (1), pp. 70-93.
- Molin P. and Ohlsson L. 1998. Points & Deviations - A pattern language for fire alarm systems. *Pattern Languages of Program Design 3*. Editors: Martin R., Riehle D., Buschmann F. Reading, MA: Addison-Wesley. pp. 431-445.
- Morris C. R. and Ferguson C. H. 1993. How Architecture Wins Technology Wars, *Harvard Business Review*, March-April, pp. 86-96.
- Nosek J. and Palvia P. 1990. Software Maintenance Management: Changes in the Last Decade. *Journal of Software Maintenance: Research and Practise*. New York, NY:John Wiley & Sons. 2 (3), pp. 157-174.
- Parnas D.L. 1972. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15 (12), pp. 1053-1058.
- Parnas D.L., Clements P. and Weiss D., 1985. On the Modular Structure of Complex Systems, *IEEE Transactions on Software Engineering*, 11(3).
- Perry D.E. and Wolf A.L. 1992. Foundations for the Study of Software Architecture. *Software Engineering Notes*. New York, NY: ACM Press. 17(4), pp. 40-52.

- Poulin, J.S. 1997. *Measuring Software Reusability*. Reading, MA: Addison Wesley.
- Prieto-Diaz R. and Neighbors J. 1986. Module Interconnection Languages. *Journal of Systems and Software*. New York, NY: Elsevier Science Inc. 6 (4), pp. 307-334.
- Randell B., 1979. Software Engineering in 1968, in Proc. of the 4th International Conference on Software Engineering. San Diego, CA; IEEE Computer Society Press, pp. 1-10
- Richardson D.J. and Wolf A.L. 1996. Software Testing at the Architectural Level. *Proceedings of the Second International Software Architecture Workshop*. New York, NY: ACM Press. pp. 68-71.
- Robson C. 1993. *Real World Research*. Oxford, UK: Blackwell Publishers Ltd.
- Rubin K. and Goldberg A. 1992. Object Behaviour Analysis, *Communications of ACM*, September, pp. 48-62
- Rumbaugh J., Blaha M., Premerlani W., Eddy F. and Lorensen W. 1991. *Object-oriented modeling and design*, Englewood Cliffs, NJ: Prentice Hall.
- Runeson P. and Wohlin C. 1995. Statistical Usage Testing for Software Reliability Control. *Informatica*. Ljubljana, Slovenia: Slovene Society Informatika. 19(2), pp. 195-207.
- Sandewall E., Strömberg C., and Sörensen H. 1981. Software Architecture Based on Communicating Residential Environments, *Fifth International Conference on Software Engineering (ICSE'81)*, San Diego, CA, IEEE Computer Society Press. pp. 144-152.
- Shaw M. and Garlan D. 1996. *Software Architecture - Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall.
- Shaw M., DeLine R., Klein D., Ross T., Young D., Zelesnik G. 1995. Abstractions for Software Architecture and Tools to Support Them, *IEEE Transactions on Software Engineering*, 21(4), pp. 314-335.
- Shlaer S. and Mellor S.J. 1997. Recursive Design of an Application-Independent Architecture. *IEEE Software*, 14(1), pp. 61-72.
- Smith C.U. 1990. *Performance Engineering of Software Systems*, Reading, MA: Addison-Wesley.
- Soni D., Nord R.L. and Hofmeister C. 1995. Software architecture in industrial applications. *Proceedings of the 17th International Conference on Software Engineering*. New York, NY: ACM Press. pp. 196-207.
- Stark G.E. and Oman P.W. 1997. Software Maintenance Management Strategies: Observations from the Field. *Journal of Software Maintenance: Research and Practice*. New York, NY: Wiley & Sons. 9 (6), pp. 365-378.
- Sutcliffe A.G., Maiden N.A.M., Minocha S. and Manuel D. 1998. Supporting Scenario-Based Requirements Engineering. *IEEE Transactions on Software Engineering*. 24(12), pp. 1072-1088.

- Swanson E.B. 1976. The Dimensions of Maintenance. *Proceedings of the 2nd International Conference on Software Engineering*, New York, NY: ACM Press, pp. 492-497.
- RAISE Language Group, The. 1995. Raise Method Manual. London, UK:Prentice Hall.
- Tichy W.F. 1998. Should Computer Scientists Experiment More? *IEEE Computer*. May, 1998, 31(5), pp. 32-40.
- Tracz W. 1995. DSSA (Domain-Specific Software Architecture) Pedagogical Example. *ACM Software Engineering Notes*. New York, NY: ACM Press. 20(3), pp. 49-62.
- Turver R.J. and Munro M. An Early Impact Analysis Technique for Software Maintenance, *Journal of Software Maintenance: Research and Practice*. New York, NY: Wiley & Sons. 6 (1), pp. 35-52.
- Wirfs-Brock R., Wilkerson B. and Wiener L. 1990. *Designing Object-Oriented Software*, Upper Saddle River, NJ:Prentice Hall.
- Wohlin C., Runeson P., Höst M., Ohlsson M.C., Regnell B. and Wesslén A. 2000. *Experimentation in Software Engineering: An Introduction*. Boston, NY: Kluwer Academic Publishers.
- Yin RK. 1994. Case Study Research Design and Methods. 2nd ed. *Applied Social Research Methods Series*, London, UK: Sage Publications.

Author Biography

**PerOlof PO
Bengtsson**



He received his Master of Science degree in Software Engineering from University of Karlskrona/Ronneby (now Blekinge Institute of Technology), Sweden, 1997 and was awarded a price as the best M. Sc. graduate in software engineering of his class.

During two years (1995-1997) of his studies he worked as a quality assurance and reuse consultant at the Ericsson subsidiary Ericsson Software Technology AB, Sweden. Between 1997 to 2002 he has been on leave from Ericsson for his doctoral studies at Blekinge Institute of Technology. In 1999 he successfully defended his Licentiate thesis.

PerOlof's research interests include software architecture/ product line architecture design and evaluation, especially methods for predicting or estimating software qualities from software architecture. He has also been a working member of the European ITEA project ESAPS. ESAPS was a project consortium with several European industrial and academic partners.

His teaching experience includes four years of teaching a Software Architecture course at the university as well as teaching a full-semester software engineering project course for three years. Both courses were on the third year of the software engineering curriculum.

On his free time PO is an active musician and has played the trumpet since he was ten years old. Should time permit he also enjoys photography.

APPENDIX A: Individual Information Form

Identification: _____

Started SE Curriculum: ☐ 1994 ☐ 1995 ☐ 1996 ☐ 1997

Working Since: _____ (Year)

Number of Study Points: _____

Maintenance Experience: _____ (Years/Months)

Software Development Experience: _____ (Years/Months)

Knowledge in:

C	<input type="checkbox"/> None	<input type="checkbox"/> Novice	<input type="checkbox"/> Skilled	<input type="checkbox"/> Expert
C++	<input type="checkbox"/> None	<input type="checkbox"/> Novice	<input type="checkbox"/> Skilled	<input type="checkbox"/> Expert
Java	<input type="checkbox"/> None	<input type="checkbox"/> Novice	<input type="checkbox"/> Skilled	<input type="checkbox"/> Expert
Eiffel	<input type="checkbox"/> None	<input type="checkbox"/> Novice	<input type="checkbox"/> Skilled	<input type="checkbox"/> Expert
Pascal/Delphi	<input type="checkbox"/> None	<input type="checkbox"/> Novice	<input type="checkbox"/> Skilled	<input type="checkbox"/> Expert
Visual Basic	<input type="checkbox"/> None	<input type="checkbox"/> Novice	<input type="checkbox"/> Skilled	<input type="checkbox"/> Expert
Assembly language	<input type="checkbox"/> None	<input type="checkbox"/> Novice	<input type="checkbox"/> Skilled	<input type="checkbox"/> Expert

Software Modelling Experiences:

Booch	<input type="checkbox"/> None	<input type="checkbox"/> Novice	<input type="checkbox"/> Skilled	<input type="checkbox"/> Expert
OMT	<input type="checkbox"/> None	<input type="checkbox"/> Novice	<input type="checkbox"/> Skilled	<input type="checkbox"/> Expert
Objectory	<input type="checkbox"/> None	<input type="checkbox"/> Novice	<input type="checkbox"/> Skilled	<input type="checkbox"/> Expert

Training in Software Architecture: _____ (Days or Credits)

Requirements Experience: _____ (Years / Months)

APPENDIX B: Individual Scenario Profile

Identification: _____

Project: ☐ Alpha ☐ Beta

Previous Domain experience: _____ Years

No.	Cat.	Scenario Description	Weight
S1			
S2			
S3			
S4			
...			
S80			

ID	Category
C1	
C2	
...	
C10	

APPENDIX C: Group Scenario Profile

Project: ☐ Alpha ☐ Beta

Group Identification: _____

Identification: _____ & _____ & _____

Previous Domain Experience: _____ & _____ & _____ Years

No.	Cat.	Scenario Description	Weight
S1			
S2			
S3			
S4			
...			
S80			

ID	Category
C1	
C2	
...	
C10	

APPENDIX D: Virtual Group Results

Table 1: Project Alpha - Virtual Group Profile Data

Profile	Score	Length	# in Ref.	# scoring/ # non-scoring	Members
102*	22	10	7	2,33	Ernie
105*	22	17	7	0,70	Ivan
104*	20	14	6	0,75	Harald
101*	15	10	5	1,00	David
100*	14	9	4	0,80	Charlie
103*	5	12	2	0,20	Frank
207	29	23	10	0,77	David, Harald
211	29	23	10	0,77	Ernie, Ivan
214	29	28	10	0,56	Harald, Ivan
210	27	20	9	0,82	Ernie, Harald
209	27	22	9	0,69	Ernie, Frank
201	26	17	9	1,13	Charlie, Ernie
205	26	17	9	1,13	David, Ernie
208	26	24	9	0,60	David, Ivan
204	24	23	8	0,53	Charlie, Ivan
213	24	28	8	0,40	Frank, Ivan
200	23	18	8	0,80	Charlie, David
203	22	20	7	0,54	Charlie, Harald
212	22	25	7	0,39	Frank, Harald
202	19	21	6	0,40	Charlie, Frank
206	18	21	6	0,40	David, Frank
315	33	35	12	0,52	David, Harald, Ivan
311	31	27	11	0,69	David, Ernie, Harald
302	31	29	11	0,61	Charlie, David, Harald
304	31	29	11	0,61	Charlie, Ernie, Frank
306	31	29	11	0,61	Charlie, Ernie, Ivan

Table 1: Project Alpha - Virtual Group Profile Data

Profile	Score	Length	# in Ref.	# scoring/ # non-scoring	Members
312	31	29	11	0,61	David, Ernie, Ivan
318	31	32	11	0,52	Ernie, Harald, Ivan
317	31	34	11	0,48	Ernie, Frank, Ivan
319	31	39	11	0,39	Frank, Harald, Ivan
300	30	24	11	0,85	Charlie, David, Ernie
305	29	26	10	0,63	Charlie, Ernie, Harald
310	29	28	10	0,56	David, Ernie, Frank
316	29	31	10	0,48	Ernie, Frank, Harald
309	29	33	10	0,43	Charlie, Harald, Ivan
313	29	33	10	0,43	David, Frank, Harald
303	28	30	10	0,50	Charlie, David, Ivan
301	26	29	9	0,45	Charlie, David, Frank
308	26	34	9	0,36	Charlie, Frank, Ivan
314	26	34	9	0,36	David, Frank, Ivan
307	24	31	8	0,35	Charlie, Frank, Harald
401	33	33	12	0,57	Charlie, David, Ernie, Harald
400	33	35	12	0,52	Charlie, David, Ernie, Frank
402	33	35	12	0,52	Charlie, David, Ernie, Ivan
412	33	38	12	0,46	David, Ernie, Harald, Ivan
405	33	40	12	0,43	Charlie, David, Harald, Ivan
407	33	40	12	0,43	Charlie, Ernie, Frank, Ivan
414	33	43	12	0,39	Ernie, Frank, Harald, Ivan
413	33	45	12	0,36	David, Frank, Harald, Ivan
406	31	37	11	0,42	Charlie, Ernie, Frank, Harald
408	31	37	11	0,42	Charlie, Ernie, Harald, Ivan
410	31	37	11	0,42	David, Ernie, Frank, Harald
403	31	39	11	0,39	Charlie, David, Frank, Harald
411	31	39	11	0,39	David, Ernie, Frank, Ivan
404	28	40	10	0,33	Charlie, David, Frank, Ivan
409	24	31	8	0,35	Charlie, Frank, Harald, Ivan
500	33	43	12	0,39	Charlie, David, Ernie, Frank, Harald
502	33	43	12	0,39	Charlie, David, Ernie, Harald, Ivan
501	33	45	12	0,36	Charlie, David, Ernie, Frank, Ivan
504	33	48	12	0,33	Charlie, Ernie, Frank, Harald, Ivan
505	33	48	12	0,33	David, Ernie, Frank, Harald, Ivan
503	33	50	12	0,32	Charlie, David, Frank, Harald, Ivan
600	33	53	12	0,29	All six.

Table 2: Project Beta - Virtual Group Profile Data

Profile	Score	length	# in Ref.	Members
100*	9	10	4	Arthur
101*	9	12	4	Bertram
102*	7	12	3	Gordon
200	11	19	5	Arthur, Bertram
201	11	20	5	Arthur, Gordon
202	11	22	5	Bertram, Gordon
300	11	28	5	All three.



ISBN 91-7295-007-2
ISSN 1650-2159