Strategies for Management of Architectural Change and Evolution

Nedstam, Josef

2005

Strategies for Management of Architectural Change and Evolution

Josef Nedstam

LUND UNIVERSITY
Department of Communication Systems
Faculty of Engineering

ISSN 1101-3931
ISRN LUTEDX/TETS—1077—SE+192P

Printed in Sweden
E-kop
Lund 2005

To mom

This thesis is submitted to Research Board FIME – Physics, Informatics, Mathematics and Electrical Engineering – at Lund Institute of Technology (LTH), Lund University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Engineering.

Contact Information:

Josef Nedstam
Department of Communication Systems
Lund University
P.O. Box 118
SE-221 00 LUND
Sweden

Tel: +46 46 222 96 68
Fax: +46 46 14 58 23
e-mail: josef.nedstam@telecom.lth.se
http://www.telecom.lth.se/Personal/josefn/

Abstract

Software architecture, the underlying structure to a software system, is an asset which can be invested in. Such investments can later be capitalized on in the form of e.g. increased flexibility and enhanced maintainability. These benefits may be gained in a system developed in one project, but are much more visible in a strategic perspective when several projects share resources. Architectural decisions must then be based not only on technical considerations, but also on organizational and business factors. Organizational factors come into play as the software development process is tightly related to the design rules surrounding the software architecture. Business factors are important, as any attempt at generating value by investing in the software architecture must be aligned with the organization's overall view on value generation. Challenges in this field lie in raising the awareness of organizational and business factors for developers, and the abstractness of software architecture from a management point of view.

This thesis has involved multiple case studies at organizations facing challenges related to these aspects of software architecture. The research has been conducted mainly from a qualitative standpoint, to build theory from empirical observations. Interviews as part of architecture and process assessments have been the primary tool for gathering information, and the results have been validated by a continuing effort to find cases that broaden the emerging theories.

Different development models linked to architectural strategies have been investigated, and the flexibility of software processes has been assessed with respect to e.g. architectural changes. Study of the implementation of architectural changes has lead to a suggestion for a process for architectural change. Architectural decisions and changes drive the evolution of a software architecture, and the thesis gives a framework for evolution of software architecture and development strategy in general. One such strategy, Open Source Software development, is analyzed in-depth. An attempt to quantify the benefit of an architectural strategy is also made, when a model for decision support on reuse is implemented in a tool.

The thesis contributes to increasing the awareness among both developers and managers of the role of the software architecture as an enabler and mediator of technical as well as business goals.

Acknowledgements

Thanks to my supervisor Martin Höst, and my professors over the years: Per Runeson, Ulf Körner, Claes Wohlin and Ross Jeffery. Thanks, Karl Cox, for inspiring me to go to Australia, and thanks again to Ross and NICTA for financial support during my time there. Thanks also to Fysiografen for such funding, and to my parents for funding my conference trip after Australia. Special thanks to Mark Staples for showing me that it's possible to do research together with other people.

Thanks to all my co-authors, and to my co-workers at SERG/Telecom and NICTA/ESE. Among these, special mention to Lena Karlsson, Daniel Karlström, Thomas Olsson, Magnus C. Ohlsson, and Enrico Johansson. Finally thanks again to my parents, and also to family and friends!

Contents

I Introduction

Software has very few laws of nature that limit its usage, compared to other engineering disciplines. A bridge builder will be subjected to laws of nature beyond her control when selecting solutions to span a bridge across some obstacle. A software developer would in a similar situation have many more degrees of freedom, and most restrictions would be constructed by man, such as limitations on the programming language used. When all these degrees of freedom are utilized, each effort to solve a problem becomes a standalone project.

The drawback of this situation lies foremost in productivity, as commonalities between solutions are not exploited. A software architecture, i.e. an enforced structure to the components of a software system, can be chosen to limit the set of possible solutions, to focus development on solution subsets which are known to have certain qualities, such as security, safety or performance. To exploit the commonalities between several solutions, the concept of software architecture must however be extended to cover several products – to treat the software architecture as an asset.

There are two main challenges to this approach. A software architecture used this way implies imposing rules on the developers, and it can be difficult to make the developers adhere to these rules. It is also difficult to make management on the business side invest in an abstract asset such as a software architecture, rather than in e.g. concrete functionality that can immediately fulfill some customer need.

This thesis contains studies that aim at focusing on the architecture as a strategic asset, at exploiting its full potential.

1. Outline and Related Work

This thesis is introduced in this chapter, were an initial motivation is given, along with a general description of the foundational concepts of software process and software architecture. The chapter is followed by a description of the research methodology, also containing the research goals and research questions of this thesis. Chapter III continues this methodology description by presenting the sample of organizations studied for this thesis. The subsequent two chapters

present surveys of related work in Software Product Lines, and the business and economics of software. The empirical studies performed during this thesis are then presented. Chapter VI presents a study on process flexibility, with results regarding the linkage between software organizations, processes and architectures. It also provides insights into scenario-based models, and on how process simulation can be used to study software processes. Chapter VII further investigates the linkage between organization, process and architecture by examining a number of cases of architectural change. A process for architectural change is given. Chapter VIII enhances this notion of architectural change to cover architectural evolution, i.e. sequences of architectural change. A framework for architectural evolution in face of changing markets is given. The framework involves several strategies for software architecture and software development in general. One additional such strategy is analyzed in Chapter IX, open source software development. Chapter X presents an effort to quantify the value of an architectural strategy for reuse, and the thesis is concluded in Chapter XI.

In the course of this research a number of papers have been publicized. The following have been used to a smaller or larger degree in this thesis:

Exploring Bottlenecks in Market-Driven Requirements Management Processes with Discrete Event Simulation

Martin Höst, Björn Regnell, Johan Natt och Dag, Josef Nedstam and Christian Nyberg

The Journal of Systems and Software, No. 59, pp 323-332, 2001

This paper evaluates discrete-event simulation as a tool to perform early evaluation of process change proposals. Simulation is one of the tools used to analyze the organizations and processes in the case study. The paper contributes to the description of line-oriented development in Chapter VI and leads up to the organizational framework presented there.

Understanding Software Processes through System Dynamics Simulation: A Case Study

Carina Andersson, Lena Karlsson, Josef Nedstam, Martin Höst and Bertil I. Nilsson

Proceedings of the 9th IEEE Conference on Engineering of Computer-Based Systems (ECBS'02), Lund, Sweden, April 2002

This paper evaluates system-dynamics simulation as a tool to perform evaluation of process change proposals. It has for this thesis increased the depth of study of the organization Company A presented in Chapter III and analyzed in Chapters VI and VIII.

Practitioners' Expectations on Software Processes

Martin Höst, Daniel Karlström and Josef Nedstam

Workshop on Learning Software Organizations (LSO 2003), Luzern, Switzerland, April 2003

This is a report from an interview study where practitioners have been interviewed about their expectations on the software process. It has contributed knowledge to the Company E case in this thesis. The organization is presented in Chapter III, used as reference for the analysis in Chapter VII, and one of the cases studied in Chapter VIII.

A Case Study on Scenario-Based Process Flexibility Assessment for Risk Reduction

Josef Nedstam, Martin Höst, Björn Regnell and Jennie Nilsson

Proceedings of the 3rd International Conference on Product-Focused Process Improvement (PROFES'01), Kaiserslautern, Germany, September 2001

This paper presents the concept of process flexibility, and a process flexibility assessment method, in a case study. The method is based on risk analysis, and the case study identified opportunities for process improvements, although no clear line was drawn between process issues and other issues such as organization and architecture. This is the basis for Chapter VI.

The Architectural Change Process

Josef Nedstam, Even-André Karlsson and Martin Höst

Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04), pp. 27-36, Redondo Beach, CA, August 2004

The paper presents a process for decision-making in architectural changes. The process is based on the functional change process used by most companies, theory of organizational change, and experiences from seven architectural changes at three companies. The paper concludes that architectural changes lead to organizational and process changes, and that these are as important as the technical parts of such changes. An additional case study has been used for validation. This paper is the basis for Chapter VII.

The study also exists in an initial workshop version: Nedstam, J., Karlsson, E.-A. and Höst, M, "Experiences from the Architectural Change Process", in *Proceedings of 2nd International Workshop From Software Requirements to Architecture (STRAW'03)*, Portland, Oregon, May 2003.

Evolving Strategies for Software Architecture and Reuse

Josef Nedstam and Mark Staples

Submitted to Journal of Software Process: Improvement and Practice, 2005

The paper presents a framework for architectural evolution, which describes how organizations can change strategies for software architecture and general software development as their market and business environment changes. The framework is developed from studies of the architectural evolution of 13 companies. This paper is the basis of Chapter VIII.

The study also exists in two workshop versions: Nedstam, J. and Karlsson, E.-A., "Experiences from Architectural Evolution", in *Proceedings of 5th Australasian Workshop on Software and Systems Architecture*, Melbourne, Victoria, April

2004; and: Nedstam, J., "Finalizing a PhD Thesis in Architectural Evolution", in *Proceedings of 6th International Workshop on Economics-Driven Software Engineering (EDSER-6)*, Edinburgh, Scotland, May 2004.

Open Source Business Models in Practice: A Survey of Commercial Open Source Introduction

Josef Nedstam, Anna Andersson and Karin Hässler

Technical Report CODEN:LUTEDX(TETS-7213)/1-14/(2005) & local 28, Department of Communication Systems, Lund University.

The paper presents a study of how nine companies have integrated open source software development into their business models. This shows an additional strategy to the paper above. The paper is the basis of Chapter IX.

A Quantitative Model for Valuation of Module Reusability

Josef Nedstam and Martin Höst

Submitted to 10th European Conference on Software Maintenance and Reengineering, Bari, Italy, 2006

The paper presents an attempt to quantify the benefits of an architectural strategy, to give one practical example of tools similar to those presented in Chapter V. The paper is the basis of Chapter X.

2. Software Products and Software Development

Theory on software products and software development is needed as a background for discussions on the importance of software architecture as a tool to improve productivity. This section discusses general attributes of software products, change as a key component for successful software development, and the evolution of software products.

2.1 Functionality and Quality

The functionality, or the feature content, of a software product is often its most visible aspect. Functionality can e.g. easily be demonstrated. Other aspects such as usability and reliability are also important, but since they are harder to specify, they are often neglected. Normally product requirements are divided into functional and non-functional requirements (Sommerville, 2001), with the unfortunate effect that all aspects other than functionality are bundled together (Bass *et al.*, 1998). Bass *et al.* instead introduce quality attributes. These are split into two major groups: those that are observable at execution, and those that are not, as shown in Table 1.

Other classifications exist, and one example is found in ISO/IEC 9126 (Sanders & Curran, 1994; ISO, 2001). That classification includes similar aspects as the one given above, but does not separate between qualities that are observable at execution or not. This thesis focuses on these quality attributes of

Table 1. Quality attributes

Observable at execution:	Not observable at execution:
Feature content	Modifiability
Performance	Portability
Security	Reusability
Availability	Testability
Usability	

software, rather than functionality. It also focuses more on the right hand side of Table 1.

2.2 Software Change

Profitable industry relies on change (Kotter, 1996). Market competition is the main driving force for this trend, accentuated by globalization of competition. New products must be released in a faster pace, new markets must be created, market shares in existing markets must increase, and costs in development, production and distribution must be reduced. This is also true for software products.

Lehman presents the notion of software evolution (Lehman *et al.*, 1997; Lehman & Ramil, 2003), to describe how a software system in operational use must change to remain of value for its users. These changes are classified as adaptive, corrective, or perfective. Adaptive changes are made as the environment of the system changes, such as a changing hardware environment, or a changing legal environment. Corrective changes are made as faults in the original system are discovered. Perfective changes are made to improve some quality attribute of the system, or to enhance the system with new features. This view on software evolution focuses on what happens to a system after it has been developed and delivered, i.e. software maintenance. This thesis broadens this view of software evolution by also looking at new development from a basis of available software assets from previous products (Rajlich & Bennett, 2000).

Software changes can be implemented on many levels of abstraction. One such class of software change is the architectural change (Sommerville, 2001). An architectural change is a structural change to the software with the purpose of changing or maintaining the quality attributes of a system, or enabling major changes to the system's functionality, i.e. adaptive or perfective change. The focus of this thesis has been on architectural changes with impact on quality attributes that are not observable at execution. The aim of such changes is to enable introduction of new products in a faster pace on the market, and decrease the costs of product development and maintenance.

Other changes of importance to the software industry are improvements of development processes (Zahran, 1998). The software process is a tool to avoid previous problems and retain experiences from projects. The quality of the software process is linked to the quality of the resulting product (Sommerville, 2001; Sanders, 1994), so software process improvement can improve software product quality, but also decrease lead-time and lower costs of software development.

2.3 Product Lifecycles

Changes, such as those described in the previous subsection, mean that software products, just as any other type of product, follow a product lifecycle on the market (Grant, 1998). A typical lifecycle of a successful product is shown in Figure 1. After an introductory phase where the product is marketed and needed infrastructure is established, the invention or product becomes desired and experiences market growth and expansion. After a while the market becomes saturated, and this is followed by the phase where a company can make the largest revenues from the, by now, mature product. Finally, introduction of other products or inventions means that this product, or the entire market, will decline.

This behavior has major implications on the software development issues studied in this thesis. The introduction will involve much innovative work, where after the structure, or architecture, of the product will stabilize. The technical choices made in this early phase will have impact throughout the lifecycle of the product, and determine much of the company's ability to develop new versions of the product in its mature, and financially interesting, phase.

The processes whereby new versions of the product are developed are also affected by the product lifecycle (Noori, 1990). As already mentioned, architectural work is important in the early stages of a product. Other factors are however also affected. Project management is crucial for successful product development in the early stages of the lifecycle, while line management can take

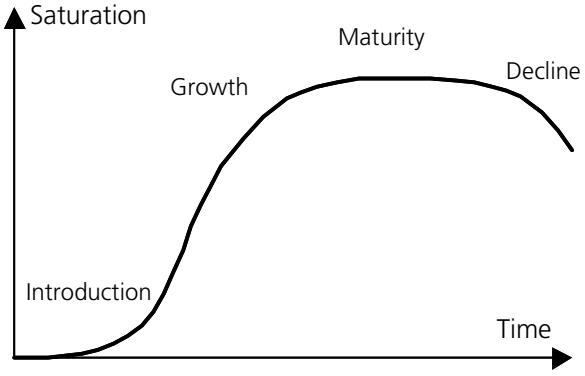


Figure 1. Product lifecycle (Grant, 1998)

more responsibility in mature products, where the organization can develop products in more of an assembly-line fashion. A similar view, the Technology Adoption Life Cycle (Moore, 1995, 1999), describing the birth and maturity of a whole market, is presented in Chapter V. It gives further guidance on the strategies available in different faces of the life cycle.

3. The Software Process

The software process refers to the activities performed when developing software, including specification, design, implementation, testing, operation and maintenance. It is sometimes referred to as the software lifecycle (Pfleeger, 1998), since it contains activities such as maintenance, which are generally not performed by the project developing the product. This should not be confused with the product lifecycle described in Section 2.3. A general definition of the word process is given in Merriam-Webster (1998):

> A series of actions or operations conducing to an end; *especially*: a continuous operation or treatment especially in manufacture

Quality management in software engineering has adopted this notion of a process from manufacturing (Bergman & Klefsjö, 2004). An improved process will result in products with improved quality. The process can also be used to share experiences within a company (Basili *et al.*, 1994), and process improvement can lead to decreased costs and lead-time (Dion, 1993).

The definition above implies that if software has been developed, that development has followed some process, even if this process has not been defined by the developing organization. A process definition, or any document describing the process, does on the other hand not guarantee that development is done according to the process (Bandinelli *et al.*, 1995). This thesis focuses on the actual work done in projects, and how to improve the way such work is done. Focus has not been put on the relation between any documented process and the process actually being followed.

The standard reference process model is the waterfall process (Royce, 1970), where development is done in a sequence of phases: requirement specification, design, implementation, integration, test and maintenance. A phase is not started until documents produced in the previous phase are approved and baselined. Since the waterfall process was suggested, iterative (Basili & Turner, 1975) and incremental process models (Mills *et al.*, 1980) have been suggested, to enable quick response to requirements changes, and lower risks of not reaching project goals.

The waterfall model resembles the stage-gate model for product development (Cooper, 1990; Cooper & Kleinschmidt, 1993), which is used for management of development projects. Projects are split up into stages with defined goals, and at each gate the project is assessed to see if it should be continued or

cancelled. The development itself does not however have to follow the waterfall process (Karlström & Runeson, 2005).

Much of the focus in software process improvement has been on the management part of software development. The leading example is the Software Engineering Institute's Software Capability Maturity Model (CMM) (Paulk *et al.*, 1993), which focuses on using quality control from manufacturing disciplines and apply to software engineering, by measuring and improving the software process. A reaction to this trend is the developer-focused, agile processes. The main example is extreme programming, XP (Beck, 1999a; 1999b), which contains a number of programming practices, such as pair programming, small releases, test first, and 40-hour week, but also a practice of on-site customer.

A number of process characteristics are defined in (Sommerville, 2001), shown in Table 2:

Table 2. Process Characteristics

Understandability	Visibility
Supportability	Acceptability
Reliability	Robustness
Maintainability	Rapidity

For process improvement purposes, and also for selecting a company that can develop high quality products, these attributes must be assessed. Several assessment methods exist, such as ISO 9000 (ISO-9001, 1994), TickIT (TickIT, 1995), CMM (Paulk *et al.*, 1993) and Bootstrap (Kuvaja *et al.*, 1994), but most are focused on management and quality issues (Ares *et al.*, 2000). Most assessment methods are therefore used to certify a process or organization, or to reach a certain maturity level. Software process improvement is mainly concerned with making a process better in a general sense, and does not focus on support for process change in a concrete business situation. An assumption behind this thesis is that a process is not good, according to any quality attribute, in it self, but must be viewed according to the context it is being used in.

Rather than quantitative process improvement, this thesis is more concerned with qualitative process changes necessitated as the business environment or software architecture of a company changes. The need for process changes driven by architectural changes is studied in Chapter VII.

4. Software Architecture

One definition of software architecture used in this work is taken from Bass *et al.* (1998):

> The software architecture of a program or computing system is the structure or structures of the system, which com-

> prise software components, the externally visible properties of those components, and the relationships among them.

This definition implies that the architecture is an abstraction of a system that describes the components of the system, and their interaction, but not the internal details of these components. Furthermore, an architecture can and will be described by more than one structure or view. However, all systems have an architecture, even though no architect has created it, and no documents describe it. To be useful as a software asset, the architecture, just as the process, therefore must be enforced. The most tangible features of a software architecture therefore often become the design rules that enforce it.

The choices made when developing an architecture should be controlled by the driving quality attributes, i.e. the subset of quality attributes such as those given in Table 1 that is viewed as most important for the product or products that shall be developed from the architecture (Bass *et al.*, 1998). Different solutions can be evaluated with respect to several of these driving quality attributes based on a number of architecture assessment methods (Dobrica & Niemelä, 2002; Abowd *et al.*, 1997), such as modifiability in the Software Architecture Analysis Method (SAAM; Kazman *et al.*, 1994), and almost any attribute in the Architecture Tradeoff Analysis Method (ATAM; Kazman *et al.*, 1998). An overview of SAAM is given in Chapter VI.

One sketch of boxes, connected with arrows, is not sufficient to describe an architecture according to the definition above. There are a number of views on the architecture of a system. Bass *et al.* (1998) suggests a number of architectural structures, shown in Table 3:

Table 3. Architectural Structures

Module	Conceptual/Logical	Process/Coordination
Physical	Uses	Calls
Data flow	Control flow	Class

These structures show different views of the system and its architecture. Since they are intricately related to one another, they are complicated to develop, change, and update. Kruchten (1995) instead presents four views that are related almost hierarchically, see Figure 2, and a fifth view, the scenarios view, to tie the four other together. Each of the four views is connected to a stakeholder: end users for the logical view, programmers for the development view, system integrators for the process view, and system engineers for the physical view.

Research in the field of architectural changes is supported by the assessment techniques mentioned above, but most literature in the field of architecture is concerned with developing architectures for new systems. One assumption behind this thesis is that most architectural work is currently done on architectures in existing products, and this thesis focuses on how to manage change in such architectures. In a sense this way of using an architecture can be seen as

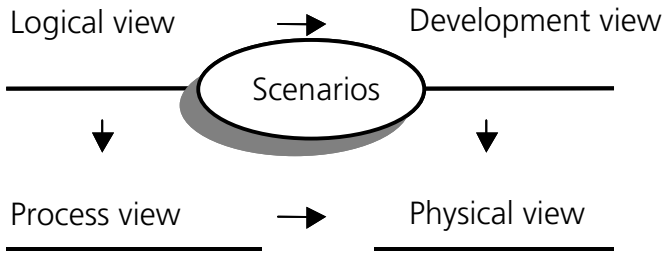


Figure 2. The 4+1 View Model of Architecture

maintenance, where defects in existing products are removed, and the products are adapted to a changing environment. The architecture studies in this thesis have however investigated how new products are developed from old products, rather than improving old products. Research in component-based software development and software product lines (Clements & Northrop, 2001) share this view, and software product line engineering is revisited in Chapter IV.

5. Relation between Process and Architecture

There are a number of similarities and related issues between software architectures and software processes (Perry, 1994). From a technical viewpoint they can both be decomposed into components that treat information, and output some type of data. This view might be useful especially when trying to automate parts of the software process. It might also be useful when trying to find strategies for decomposing a software process into activities. A problem with this view is that while software components can be controlled and behave deterministically, the teams and personnel in a software project present social and cultural behavior and challenges.

This thesis has mainly studied two other similarities. First of all, both development processes and software architectures imply rules on how software development shall be carried out at an organization. A software process exhibits this in an obvious way. The architecture is however also often seen as a set of rules, from a developer viewpoint. Secondly, the architecture and the process are both used to transfer knowledge and experiences from one project or product to another.

II Research Methodology

Settling on research questions is the first step of a successful research project. Design of a study can be done from this basis. First of the choices to then be made is to select a research approach. This will guide selection of data collection methods, and later, analysis techniques.

This thesis presents on empirical research, where conclusions are drawn based on observation of a real-world phenomenon. This section first of all presents the research goal and the research questions for the thesis. The choice of a qualitative or a quantitative approach is then discussed, and the predominately qualitative data collection and analysis methods used in this thesis are described.

1. Research Goals

The research presented in this thesis has been guided by a vision to enhance developers' and architects' abilities to improve the software architecture that supports them, so it fits better with the business goals of their organizations. Special interest has been placed on using software architecture as a tool for achieving reuse-related quality attributes.

The benefits gained by a software architecture are assumed to be context dependent, i.e. one form of successful architecture improvement is hard to transfer to another software developing organization. This means that the concrete conclusions about the specific architectures found in this thesis are hard to transfer, but that the methods and tools that have been assessed can be used in other contexts.

1.1 Goals

The vision above has been specified in a research goal, to set the constraints for this PhD Thesis:

> Help developers and architects utilize and evolve their software architectures to better support the business goals of their organizations, with respect to quality attributes related to increased productivity.

The goal has a target audience of developers and architects. A secondary goal is therefore to give the third stakeholder, management, insights into what can be achieved by a software architecture with respect to productivity. The software architect role is also deliberately vague as it will have many different names in different organizations. It should be viewed as a role with technical responsibilities not connected to one single development project, but rather to software assets common to many parallel and consecutive projects. In order for the two roles, developers and architects, to make the architecture support business goals, these business goals must be understood. Finally, quality attributes relating to productivity are such as reuse, modifiability, and extensibility. The opportunities to use the software architecture to support quality attributes such as security, safety or performance are not part of this thesis.

1.2 Questions

The goal above has been broken down into research questions, which have been explored in research projects in the scope of the thesis. These questions are as follows:

Question 1: What is the linkage between architectural strategies and business goals?

Question 2: How does the software architecture relate to the organization and the software process?

Question 3: How do organizations carry out architectural changes, and how can this process be improved?

Question 4: Which architectural strategies exist, with bearing on business goals related to productivity?

Question 5: How do software architectures evolve to continue supporting business goals in an evolving environment?

These questions have been formulated as concrete research questions in the various studies described in the chapters of this thesis. The research questions presented here has been the focus of different studies as seen in Table 4.

2. The Methodological Approach

Empirical software engineering is a relatively new field of study, and methodologies are still being established. Much empirical software engineering research has followed the quantitative tradition, as is done by Wohlin *et al.* (2000), perhaps because most researchers come from the natural sciences, where controlled experiments are familiar. Before carrying on with a research project, one should however ask oneself according to which research tradition it lends itself.

The positivistic school, using quantitative studies, has its roots in the natural sciences where universal and objective laws of e.g. physics and mathematics can be found (Kolakowski, 1972). A theory is used to form a hypothesis, which is tested through a controlled experiment on a phenomenon, separated from its context. The method should be as formal as possible, ensuring that the results will not be affected by e.g. researcher bias or any other source of subjectivity.

Table 4. Relation between research questions and papers

Question	Chapters	Linkage
1	IV, VIII, X	Business goals of software organizations are shown in Chapter IV, linkage between architectural strategies and business goals is explored in Chapter VIII, and benefits of a specific architectural strategy are analyzed in Chapter X.
2	VI, VII	Chapter VI analyses the relation between the flexibility of a software process, the software organization, and its architecture, while Chapter VII investigates process and organizational factors that have to be understood and overcome in order to change an architecture.
3	VII, X	Chapter VII presents an architectural change process based on a number of studied architectural changes, while Chapter X gives an example of a quantitative model for decision support in such a process.
4	VIII, IX	Chapter VIII presents a framework of architectural and development strategies to exploit opportunities for reuse, appropriate for a company's specific position and environment. Chapter IX investigates one additional strategy in depth.
5	VIII	Chapter VIII also shows linkage between the evolution of a business along with the evolution of the software architecture on which the business bases its offering, be it products, services, or a hybrid of the two.

The success of such methods within the natural sciences has lead to widespread use in the social sciences as well (Patton, 2001).

On the other side the qualitative school includes theoretical traditions and orientations such as ethnography, phenomenology, hermeneutics and such, all focused on the social sciences. The methods used here are not deductive but inductive, used to build theory, so-called Grounded Theory, meaning that the theory is grounded in the empirical world. In this world everything is assumed to depend on its context. Therefore, it is impossible to lift something out of its context into an experimental setting, and objectivity, whether it exists or not, is impossible to reach. Where the quantitative researcher is restricted to generalize to the analyzed context only, i.e. the population that the sample was taken from in order to represent the population, the qualitative researcher still tries to say something general about the nature of the phenomenon under study, even though a single case and a single and unique context has been studied. A qualitative researcher sees the results from a case study as one small part of a hologram, which still contains the whole picture of reality, but only from one perspective.

The choice between the two approaches is often and successfully done pragmatically. Most commonly, the qualitative approach is used in an exploratory setting, trying to create a model or a proposal for a theory, which later can

be confirmed in quantitative studies. The approaches are also frequently mixed, e.g. by using a set of qualitative interviews to put words to the numbers produced by a broad and quantitative questionnaire study.

In this thesis, the work mostly follows the qualitative approach. The studies have explored largely unchartered phenomena and social constructs. They have still however involved both collecting and generating numbers. In Robson's terminology (2002) all studies included here can be classified as case studies, although some of them possibly could have been extended to grounded theory studies.

Robson (2002) also introduces a spectrum of approaches to problem solving, ranging from pure basic research to researchers as consultants. The studies presented in this thesis have all been performed in close cooperation with industry, and the studies have aimed at generating results that can be applied by the studied companies. Most of them are therefore focused on applied research, and to some extent provide results that are directly applicable to industry. Some of the studies, such as the one presented in Chapter X, have been performed similarly to action research (Robson, 2002), where the aim is not only to broaden the boundaries of science, but also to help the subjects under study with a specific problem, and broaden the knowledge of such practitioners in relation to that problem. Action research also involves change, and as this thesis shows, change is difficult. Change driven by action research should be allowed to take years (Fullan, 1991). Even though a PhD thesis does take years, you are seldom given the opportunity to pursue one single case for lengthy periods of time.

3. Data Collection Methods

Several data collection methods are available for qualitative studies, but the most common are observations, interviews and archival studies. The following discussion focuses on interviews, as it has been the most frequently used instrument in the studies presented in this thesis.

3.1 Observation

Observation is, according to many qualitative researchers, the best method of data collection (Patton, 2001), as it gives the best opportunities for the researcher to penetrate a phenomenon. The largest critique against observations is that the observer is biased, that the observer affects the phenomenon under observation, and that it is too difficult to record what is being observed, i.e. two observers will tell a quite different story after watching the same phenomenon (Robson, 2002).

The observer bias is handled by acknowledging that all methods have researcher bias. The results will therefore present but one, or possibly a few, perspectives of the phenomenon under study. The observer's effect on the phenomenon can be handled by being an invisible observer. This can be hard to achieve or even raise ethical aspects. Another solution is participatory observation, where the observer to a varying degree participates in the phenomenon.

Even though it is often advantageous to participate fully, one must be able to use the gathered experiences in research and present the results. Therefore, it is often useful to take a step back and just be a bystander from time to time, in order to observe the phenomenon through a research viewpoint. A helpful ability or technique for this is introspection, in order to analyze what one really has been participating in and observing. The drawback of participant observations is that it demands a lot of resources.

Observation is resource intensive, as it takes weeks to blend in into a software engineering work environment. Observation has therefore only been used in specific settings, such as to better understand architectural assessment methods, and during meetings, to understand the architectural decision process in Chapter VII.

3.2 Interviews

Within interviews, as in observations, the researcher is the instrument for data collection. Since the focus is on qualitative studies, the research is often based on a loose design and not on formal methods. This means that a lot is required from the researcher in terms of both social and scientific abilities. Becoming a good interviewer requires a lot of practice. Several types of interview techniques exist, ranging from open to structured (Minichiello *et al.*, 1990). Their characteristics are shown in Table 5.

The interviews performed in this work have mostly been in the form of an interview guide, where a prepared set of issues is discussed, and the interviewees are able to introduce other issues. In some of the studies, only one issue has been discussed. In other studies, as in Chapter VI, follow-up interviews have been performed following a structured interview technique, where the participants have mostly been presenting quantitative data about their organizations and processes.

There are different ways of ensuring a valid description of an interview. In structured interviews and questionnaires it is trivial to achieve a valid description. In open interviews it can however be difficult to account for what sometimes amounts to hours of discussions. Tape and even video recordings can be used and later transcribed. Taking notes is also an option. It is however far from obvious which is the better way. Glaser (1992) dislikes recordings, and even argues against taking notes during interviews, as it interferes with the discussion. Recordings may hinder a free and open discussion, and transcriptions require much resources, often twice as much time as the interview itself. The initial interviews of this thesis were recorded and transcribed, but later interviews were just described in keyword notes. The description validity was then assured by letting interviewees give feedback on interview summaries.

Other methods of data collection can also be used in qualitative studies. The most important is analysis of written documents, which has been used to a varying degree in most of the studies performed in this work. Literature is viewed as any other data source in qualitative studies, and just as theory emerges from a

Table 5. Types of interviews

Type	Approach	Purpose	Questions	Features
Informal conversational interview	Qualitative	Enter perspective of interviewee	Discussion around one topic led by interviewee	High resources, high dependence of interviewer
Interview guide	Qualitative	Enter perspective of interviewee, with opportunity to compare to others	Several topics given by interviewer, discussion led by interviewee	High resources, less dependence of interviewer
Standardized open-ended interview	Qualitative	Enter perspective of interviewee, with opportunity to compare to others	Questions are given word-by-word, interviewee may answer in own words	Low resources, low dependence of interviewer
Structured interview, questionnaire	Quantitative	Drawing general conclusions about phenomenon and not conclusions about interviewees	Questions are given word-by-word, on paper or verbally, interviewee is forced to answer according to fixed dimensions	Low resources, minimal dependence of interviewer

qualitative study, so does the need for literature. There is however a need for a balance between reading up on a subject before a study, and keeping an open mind to any emergent theory found in the data produced in that study.

3.3 Sampling in Qualitative Studies

Within qualitative studies the subjects are not sampled randomly, but instead purposefully. One way of doing such a sample is to find the interviewee that is likely to give the most interesting information about the phenomenon, perform the interview and do an analysis. From this analysis one tries to find a new interviewee that probably will give a totally different perspective on the phenomenon. This is repeated until no more information is extracted through further interviews (Glaser, 1992).

It is important not to think in statistical terms when selecting a sample size in qualitative studies. Whatever size selected, the results of any quantitative analysis from data collected from a qualitative, purposeful sample cannot be statistically significant, as the sample has not been selected to represent any lar-

ger population. In many of the studies presented here, as varying viewpoints as possible have been interviewed, However, it has often not been feasible to do iterative interviews. One exception is Chapter IX, where snowball sampling has been used. Here the participants are asked to name others who might be familiar with the phenomenon under study. This was repeated until new participants were naming persons who already had been interviewed. When opportunity has arisen, counterexamples or examples that have broadened theory have however been included, such as in Chapters VII and VIII.

4. Analysis in Qualitative Studies

The results from a qualitative study are not simply the transcribed and condensed interview material. This preparatory work is necessary but must be followed by a proper analysis of the data.

In order to draw conclusions from the data one must find dimensions within the phenomenon in the data, and try to find patterns among them. Dimensions are found by coding, categorizing and analyzing the data (Patton, 2001). It is of importance to compare the coding schemes and categories against the original data to see that the whole has not disappeared into a set of details.

Abstract relationships can be synthesized from the dimensions, by arranging the data into matrices according to the dimensions (Glaser, 1992). The process has to be iterated to find all of the important dimensions. Patterns that are found should be questioned, e.g. by changing perspective. By letting several researchers create their own dimensions and matrices the validity of the findings has been strengthened.

5. Assessments as a Tool in Research

Assessments or evaluations are methods for a researcher to influence the environment, the first step to perform changes and have impact on the environment. The purpose of evaluations is to assess the effects or effectiveness of the phenomenon under study, so it is not really a distinctive research strategy (Robson, 2002). Since evaluation research assigns a value to the object under study, which can include persons or companies, such research is more sensitive. It is however useful for decision support when making changes.

Several types of evaluations or assessments exist (Scriven, 1991; House, 1980). Formative evaluations are used to guide the development of a phenomenon. Summative are used to assess effects and effectiveness of a phenomenon. As an example, ATAM (Kazman *et al.*, 1998) can be used formative when developing or improving an architecture, or summative when selecting among commercial off-the-shelf components.

Evaluations can also be classified into outcome evaluation and process evaluation. The outcome of software development is products and revenues. Process evaluation therefore seems more natural when assessing software processes and architectures. However, the architecture might be viewed as an out-

come of the early activities of a software process, and the specific process for developing the architecture might be difficult to examine.

Another dimension of evaluation is whether the evaluation only produces one number indicating the monetary value of e.g. an architecture, or whether this value is assessed for different views and stakeholders, or even in qualitative terms. The bottom line is always said to be the most important factor, but Farbey *et al.* (1993) have studied how different evaluation methods, along this dimension, has been used in IT industry.

In Chapter VI, a process flexibility evaluation method has been developed and tested in a case study. The method was developed by adapting SAAM (Kazman *et al.*, 1994) to processes, since SAAM focuses on architecture flexibility. The adaptation was done with help from checklists found in Scriven (1991) and Ares *et al.* (2000). Architecture assessments according to ATAM and according to a project-centric approach were made as a first step during the work with Chapter VII.

6. Validity in Qualitative Studies

In a quantitative setting, where a phenomenon has been lifted out from its context and a controlled experiment has been performed in a laboratory, it is possible to perform a direct replication to confirm the validity of the study. In a flexible design study, such a replication is impossible, since the context of the study is always present and always changing, and cannot be recreated. Therefore, other procedures for ensuring validity and trustworthiness are needed. Robson (2002) defines three types of understanding in qualitative research, namely *description*, *interpretation*, and *theory*. These types have particular threats to their validity.

After an observation or interview, the researcher must be able to provide a valid *description* of the events. Typical ways of ensuring such validity are to take video- or audio recordings, and writing high-quality notes. The *interpretation* of what has been observed or discussed can be made invalid if the researcher imposes a preconceived meaning or framework onto the material, instead of developing such a framework or modifying any prior, suitable framework. Finally, the *theory* developed from the study can become invalid if the researcher does not consider alternative interpretations or explanations for the results. The solution here is to try to find material that does not fit the theory emerging from the study.

Glaser (1992) finds a grounded theory study valid if it has been performed as a continuing search of evidence which disconforms to the emergent theory. This ideal has been followed within reason. The participants of the study in Chapter VI were given the opportunity to change the framework of the software development environment which was emerging; one additional case was found for the architectural change process of Chapter VII, which showed the bounds of its applicability; and the framework of architectural evolution in Chapter VIII is still expanded as new cases are found.

Other threats may be related to the effect a researcher might have on a setting, interviewee bias, or researcher bias. Several techniques exist for countering these threats. A participatory observation that carries on for an extended period of time might mean that the observer blends in better with the setting. Different forms of triangulation can be used, by using many observers, many data sources or many analysis methods.

Results in the papers included in this thesis have generally been validated through repeated interpretation by several researchers, and by letting the companies and practitioners under study give feedback and review the findings. The specific procedures for ensuring validity are described in each study. The main strength in the validity of the conclusions of this thesis lies in that over 20 software organizations have been studied. The priority has been to find new cases describing similar phenomena, rather than investigating the same case from several viewpoints, such as using several respondents in the same organization, or doing lengthy observational studies. The following chapter introduces the case companies, in order to give a fair view on the context wherein conclusions can be generalized.

III Context of Studies

A discussion of the context of the presented studies can provide insights of how to transfer the results of this thesis into industry, both by showing in what situations the knowledge can be directly transferable, and by indicating when suggested guidelines and methods might have to be adapted to fit a new context. Under the assumption that such technology transfer is complicated, as stated in the research goal, Chapter II, information about the context of the findings in this report is crucial to gain any type of generalizability.

1. Studied Organizations

Table 6 gives an overview of the 23 studied organizations. The organizations are anonymized, because a number of them volunteered information on condition of confidentiality. *Domain* indicates the application domain, or the type of products the organization is developing for its customers. *Size* is a measure of the number of software developers employed by the organization. Small means not much more than 10, medium not much more than 100, and large above 100. Some organizations have however been significantly larger, but Table 6 gives the size of the organizational unit under study, rather than the size of the multinational companies which are included. This metric could also be based on revenues, and the two would often be related if the main line of business in the company would be software development. This is however not the case for several of the companies under study. *Focus* therefore shows if the company mainly focuses on hardware or software, and if the company primarily develops products or provides their customers with services. The platform option is an in-between, where the company supplies an incomplete product which needs services to be customized and installed. *Scope* indicates whether this offering is directed at a mass market or at individual customers, and is somewhat linked to the type of offering. The individual organizations are described in the following, with a varying degree of detail. Information specific to certain research issues are presented in the thesis chapter treating that issue.

Table 6. Studied Organizations

Company	Domain	Size	Focus	Scope
A	Consumer electronics	Large	Hardware product	Mass market
B	CASE tools	Large	Software product	Niche market
C	Consumer electronics platform	Large	Hardware platform	Customer specific
D	Control systems	Large	Hardware/software platform	Mass market
E	Radar systems	Medium	Hardware product	Customer specific
F	Consumer electronics	Large	Hardware product	Mass market
G	Telecommunications application platform	Medium	Software platform	Niche market
H	Newspaper management system	Medium	Software product	Niche market
I	Webshop solutions	Small	Software platform	Niche market
J	Information system development tool	Small	Software platform	Customer specific
K	Biomedical analysis tools	Small	Hardware product	Niche market
L	Healthcare services	Large	Information system	In-house
M	Combat simulators	Medium	Hardware/software product	Customer specific
N	Internet payment solutions	Medium	Software product	Niche market
O	Cash machines	Small	Hardware product	Niche market
P	Embedded web-server products	Medium	Hardware product	Mass market
Q	Embedded real-time systems	Large	Software service	Customer specific
R	On-demand information system services	Large	Software service	Customer specific
S	Telecommunications	Large	Hardware/software systems	Varies
T	Development tools for embedded systems	Small	Software service	Niche market
U	Embedded operating systems	Medium	Software product	Niche market
V	Telecommunications services	Large	Software service	Customer specific
W	CASE tools	Large	Software product	Mass market

1.1 Company A

Company A developed consumer electronics. The individual products were based on a so called application platform. The application platform development process used at Company A was assessed for the study of Chapter VI. The purpose of the platform was to serve as a basis for several products released within a time-period. The products were developed in parallel, and the first product to be based on a platform was developed simultaneously as the platform. The first product therefore had a large impact on the requirements of the entire platform. A platform contained 90 - 95% of the code of a product, and most of the code in a platform was reused from previous platforms.

The project studied in Chapter VI was the second platform project to result in a product for the public market, and therefore the details and interactions of the platform and product processes were still taking form. The project was selected because it was in its requirement specification phase, the part of the process that was most mature.

During this phase the market requirements were first analyzed and refined to requirements for the various user functions. They were then decomposed into module requirements. The step from functional to module requirements involved design work, as the requirements were mapped onto an existing architecture. The work during the requirement phase was organized in 15 functional groups, each responsible for a well-defined functional area. Those groups included the roles responsible for product management, requirement specification and software design and their tasks were to analyze, document and prioritize the requirements and also to make sure that there was compliance between the market requirements and the functional requirements. When the requirements were in baseline, the project was reorganized into module teams that should implement the functions. During this transition an integration plan was made to determine which functionality each module should contain at the end of each development increment. After implementation the modules were tested and integrated, while the integration and system test was at the time the responsibility of the projects developing products from the platform.

After this study, Company A was split in two companies, Companies C and F, along the interface of the platform. This dynamic is described in Chapter VIII.

1.2 Company B

Company B develops software engineering tools. One of their main products is a CASE tool that consists of a front-end with editors for various types of diagrams and source code, and a back-end for compiling the diagrams into code. Other utilities such as a simulation tool are also part of the development tool.

Company B is used as an example in the discussion section of Chapter VI. At the time that study was made, Company B released a new version of their product every six months, and successive development cycles for these releases overlapped. Features were implemented by development teams in an assembly-

line fashion, described by Regnell *et al.* (1998; 2001). A shift in strategy was eventually necessitated, and this is discussed in Chapter VIII. The changes leading up to this shift in strategy are analyzed in Chapter VII. The organization has architects per project but no established line organization for architecture, and module responsibility is assigned to senior experts.

1.3 Company C

Company A was split in two companies. Company C is the half that develops platforms for consumer electronic devices. These platforms are sold to external customers who configure the services within the platform to create complete products. The software platform consists of a number of modules, and a middleware layer hides the internal architecture from the customers.

Projects are organized in: a project management group, with product management responsibility; a system-engineering group, with expert groups and function groups responsible for major features within market requirements; and a system realization group, which receives specifications from the system engineering group and develops the platform. The system realization group is divided into a hardware- and a software branch, which are subdivided into development teams, each responsible for a set of modules. The organization has defined roles responsible for each module. These persons work with function groups during specification, and development teams during implementation. The company also has a dedicated architecture group that performs most of its work within projects, especially supporting and influencing the system-engineering group.

1.4 Company D

Company D develops control system environments for industrial automation, e.g. chemical plants, dairies, etc. The control system environment consists of both a development view, called control builder, and a deployment view, i.e. the controller itself. Within the control builder, controllers can be designed by specifying hardware sensors and actuators, constructing control loops, and connecting variables in those control loops to the hardware devices. A fully specified system can then be compiled and deployed onto a controller in a control system.

Company D typically carries out one large project at a time, involving the entire organization. Each project evolves the same product further by adding features to the control builder, e.g. new editor facilities, and the controller, e.g. new hardware interfaces. Implementation proposals are developed during a feasibility study. Accepted implementation proposals pass a tollgate, in line with the stage-gate approach (Cooper, 1990; Cooper & Kleinschmidt, 1993), after which implementation begins. Development is organized into teams, each working on a number of implementation proposals. Work is feature-focused and the organization has no module-responsible and no architects, but instead

relies on senior developers to take responsibility for long-term architectural goals.

1.5 Company E

Company E develops radar systems for various types of installations. Each system has been developed in its own project and for one customer. These projects have spanned several years, and the organization has focused on one customer and product at a time. The projects have followed an iterative development process, but have had little external controls during the main development parts of the projects. As the organization gains domain knowledge the customer requirements on some radar products are better understood. These mature product types can therefore be sold to new customers without large development projects. The organization also has opportunities to extract functionality general to all its products for reuse.

1.6 Company F

Company F is the other resulting half of Company A, after the split mentioned in Section 1.3, and is therefore developing consumer electronics from a hardware/software platform provided by Company C. The developed products are all belonging to the same specific application domain, and the platform is also specific to this domain. The company develops some 20 products each year – some with small variation such as internationalization, while some vary to a greater extent, such as by focusing on high-end or low-end market segments. The market pressure on this organization is therefore high. The organizational structure and processes are similar to those of Companies A and C.

1.7 Company G

Company G started out as a consultancy business in services for the public switched telephone network, and has packaged their domain knowledge in an application server. They work in close cooperation with companies providing gateways between telephone systems and the Internet. Depending on their investors they shift focus from developing a generic application server for several customers, to developing specific solutions with custom applications for one customer.

1.8 Company H

Company H develops a newspaper management system with some 80 applications such as subscriptions, advertisements and so on. They introduced a platform for graphical user interfaces to enable portability between Mac and Windows. Versions of the system were generated for individual customers by parameterization, but their solution did not allow for much variation. Problems with differing customer demands were instead solved by a customer forum were the customers often managed to agree on their requirements for the system.

1.9 Company I

Company I worked as a consultancy business, where each project resulted in a web solution customized for one customer. Similarities between projects led them to develop a wizard-style tool for setting up a web shop. This was to be sold along with services such as tailoring and development of specific applications and components.

1.10 Company J

Company J was a startup which developed a tool for developing information systems. Views on business information could be generated automatically, while business logic had to be implemented for each customer. Such development was supposed to be done by the customer, but the company was initially funded by consultancy services in information systems, often to do such development.

1.11 Company K

Company K develops tools for automatic blood analysis, specific for each customer. They extracted generic parts into a platform, to be able to cut lead-times for new customers. They however saw problems in generalizing user interface parts, as customers often had quite different requirements on user interfaces. From this platform they also tried to broaden their market scope by developing bone marrow analysis tools.

1.12 Company L

Company L is a healthcare organization, composed of several hospitals and primary health care units. Each regional organization has had its own software systems, and many of them have also developed their own software systems. Such development is now in the process of being outsourced, and the various systems are being consolidated. The overall goal is to be able to let patients have electronic journals which can be used in the same way by the various healthcare facilities in the region.

1.13 Company M

Company M develops combat simulators, and also some civil defense simulators for e.g. firefighting. Some simulators are software-only, while others are embedded in actual, or training, weapons systems. A complete customer solution may consist of many interconnected simulators on several tactical layers. Company M has used a common architecture for all products, and has tried to maintain generic components which are to be used as-is by projects developing customer solutions. The challenge lies in keeping components generic, even though customer projects at their final stages often just “clone-and-own” components (Clements & Northrop, 2001), i.e. take a copy of the source code and develop in their own direction without regard for the needs of other projects.

1.14 Company N

Company N develops credit card payment solutions for Internet. They started out by a baseline, or reference implementation, which customer projects have been free to modify to come up with a complete solution. As a considerable volume of new customer demands has emerged, a new baseline was warranted, but it was very labor intensive to lift all products up to this new level of functionality, as the customized code based on the previous baseline was hard to identify and disentangle from generic code. They have since started implementing a light-weight software product line. These efforts have been described by Staples & Hill (2004) and are a part of the analysis in Chapter VIII.

1.15 Company O

Company O is a company offering cash processing solutions such as coin sorters. The organization has 400 employees and software development at Company O is done by approximately eight developers organized as a unit within the R&D department. Company O has introduced an agile development process (Beck, 1999a; 1999b) along with open source software development (Raymond, 1998).

1.16 Company P

Company P was founded in 1984, and develops network cameras and video servers, servers for printers, scanners and storage devices as well as wireless access points for mobile connection to local networks and the Internet. Their approach to software product lines has been studied by Bosch (2000).

1.17 Company Q

Company Q develops integrated hardware and software solutions for embedded real-time systems, and has started using open source software. Their solutions are sold to individual customers as services rather than products.

1.18 Company R

Company R has in recent years shifted its main focus from providing hardware products to providing software services and solutions for business. Their open source software strategy is further described in Chapter IX.

1.19 Company S

Company S is one of the world's leading organizations within telecom industry. Some of the other organizations described in this thesis are part of Company S, and their overall strategy to open source software is further described in Chapter IX.

1.20 Company T

Company T has 30 employees and works mainly as a distributor of development tools for embedded systems.

1.21 Company U

Company U provides customers with a Linux distribution for embedded systems. Company U offers customers what is called a subscription to the Linux distribution, where an annual fee is paid for upgrades and support. All source code is provided. Customers are mainly from the consumer electronics market and the mobile communication market, but there are also some customers from the automation industry. Company U uses Eclipse technology for the in-house developed IDE DevRocket.

1.22 Company V

Company V has 3000 consultants, which operate in four different areas: Mobile Devices, Products, Operators & Networks and Enterprises & Industry. Their strategy regarding open source software is further described in Chapter IX.

1.23 Company W

Company W developed the first Integrated Design Environment (IDE). Today focus is on all activities in the software development process, and besides IDEs like JBuilder and Delphi, Company W offers software tools for requirement management, configuration management, build systems and design to their customers.

2. Scope and Features of Studied Cases

As this is a thesis in software engineering, the studied companies have mainly been software developers. Most however also develop hardware, or have to take hardware into special consideration. One company that ceased software development has also been studied for Chapter VIII, and one which only acts as an open source software distributor, for Chapter IX. Three companies no longer exist. Company A was reorganized and split into Companies C and F, a move which is studied in Chapter VIII. Companies J and T were trying to become product rather than service businesses, but have gone out of business.

Along that dimension – products vs. services – the thesis involves companies from those who develop shrink-wrapped mass market software, those that develop one-off products for one customer at a time, to those that provide either their technology or simply their know-how as services. End-users and customers vary from government and heavy industry, telecommunications operators and software developers, to private consumers; and products have been both embedded, information systems and off-the-shelf products.

For the companies where the software process has been studied in-depth, it has been clear that most development is project driven – projects to complete a mass-market product, or a specific customer solution. This full focus on individual projects has however had to be reconsidered for the companies that have wanted to manage and maintain architectural assets for reuse between projects. One company that had no real projects was Company B, where new products were pushed out regularly in a line-oriented model of development. Other organizations have used similar line-oriented development for reusable assets, while others have developed new releases of such assets in internal projects. Roles responsible for each software module have often been appointed. Such responsibility has sometimes been shared by introducing roles responsible for features, or user functionality. In some cases no module responsible have been assigned, but responsibility for the code has rather been shared between all developers. Project lead times, or times between product releases, have varied from months up to several years.

In the terms presented in this section, Chapter VI includes a study of the flexibility of the development process used in a project to develop an internal, reusable asset. The study shows the tight coupling between organization, architecture and process, and is contrasted with conclusions from a different model for development, line-oriented development. Chapter VII studies the process for making decisions about, and changing the software architecture. The context has varied, as to whether these changes have been implemented in product or customer oriented projects, internal projects, or in a line-oriented context. Chapter VIII takes a more external perspective, and discusses whether these organizations have a focus on individual customers, market niches, or mass markets, and how their strategies for development and software architecture are affected by these choices. Chapter IX studies how one such strategy, open source software development, can be exploited in various such business contexts. Finally Chapter X takes us back to the technicalities of software development, where introduction of some line-oriented elements must be validly motivated in an organization which is strongly driven by projects. This motivation is done by evaluating an architectural strategy in quantitative terms, i.e. by finding the value of architecture.

Before the studies made during this thesis are presented, the reader is introduced to one technique to increase productivity through reuse – software product-line engineering. Chapter V, on the business of software, gives the external business context for the internal development strategies which are studied in Chapter VIII. Chapter V also shows the current state of economics research in software engineering, such as tools to understand the value of the different quality attributes which are sought for in this effort to improve our ability to manage software architecture.

IV Understanding Software Product Line Engineering: Available Strategies and Approaches

Interest in gaining the benefits of a product line approach has risen in software industry lately. Such benefits are for instance increased productivity and mass customization. Several approaches for the introduction and evolution of a Software Product Line are presented in this literature survey, and their differences are identified. One such large difference is whether investments in general should be made up-front or as late as possible. Software Product Line Engineering still relies on company champions and pioneers, and needs to be packaged in tools and methods to be accessible to software industry in general.

Software product line engineering has attracted attention in recent years, due to its opportunities of exploiting reuse between the products in a company's portfolio. A software product line is a way to manage reuse of assets common to a set of products developed by one company, and to manage the variation between these products.

While the development of computer hardware still abides by Moore's law (Moore, 1965), software has not kept up with this pace. Software reuse has since the late sixties been seen as one method of bridging this gap. Many technical solutions to reuse have been proposed since then, including subroutines, modules, objects, and components (Karlsson, 1995; Krueger, 1992). Supporting technologies for reuse have also been developed, such as component libraries and configuration management systems.

The initial ambition of industry-wide reuse has resulted in research into COTS components. Industry has however gradually realized that libraries of general components do not suffice, but that reuse should rather be domain specific (Bosch, 2000). This means that reusable components must be adapted for the specific company developing products from them, and adapted for the specific type of applications being developed. A further problem with reuse that so far has been neglected is the managerial and organizational impact of a switch to reuse-oriented development. Resources have to be spent on reusable assets,

and the organization needs to introduce roles and structures that plan the development, use and maintenance of these assets. Software Product Line Engineering, which is an architectural approach to reuse, focuses efforts on the domain at hand, and may also support organizational adaptation. This report presents the benefits of Software Product Line Engineering, and the strategies and approaches available in literature.

The following section describes *why* Software Product Lines have become a current issue, while Section 2 includes definitions of *what* Software Product Lines are. Section 3 gives an overview of the most common frameworks for Software Product Line Engineering, and Section 4 and 5 focuses on *when* and *how* to introduce and evolve a Software Product Line.

1. Reasons for Software Product Lines

Krueger (2003) makes the case for a Software Product Line (SPL) explicit, when saying that the objective of an SPL is to optimize software engineering efficiency by exploiting commonalities among products. Software Product Line Engineering (SPLE), also called Software Product Family Engineering, is therefore the development and maintenance of several products simultaneously, where the products have enough commonality to warrant introduction of managed reuse and variation between them.

Krueger's case for SPL is broken down into more specific benefits by the Software Engineering Institute (SEI) at the Carnegie Mellon University (Clements & Northrop, 2001), which states that SPLE can help an organization:

- Achieve large-scale productivity gains
- Improve time to market
- Maintain market presence
- Sustain growth
- Improve quality
- Increase customer satisfaction
- Achieve reuse goals
- Enable mass customization
- Compensate for lack of software developers

These stated benefits are quite interrelated, and some are indirect benefits. For example, increased customer satisfaction would come from improved quality or mass customization, while market presence is maintained by improved time to market. For any company wanting to adopt SPLE, these benefits should be concretized and related to the company's business goals. SEI presents a good example of this in their Cummings case study (Clements & Northrop, 2001), where one business goal was to move their high-volume automotive diesel engines into the low-volume market of industrial diesel engines. SPLE's ability to

enable mass customization allowed them to develop products for these low-volume markets.

2. SPL Definitions

A range of similar definitions exist in the marketing field (Pride & Ferrell, 2003; Beckman & Rigby, 2003). In these definitions one company sells a range of product types, each of which can be considered a unit with regard to marketing or technology. A Software Product Line is more than just a product line, and different from a family of products that work together. A company can market a product line, but if the set of software products are not developed from a common set of core assets, with managed evolution and variation, it cannot be called a Software Product Line. A family of software products will often use a common asset, but will not be a Software Product Line if there is no managed variation between the products. Key components of SPLE are therefore reuse and variability.

Bosch's view on SPL (2000, p162) has its focus on reuse between members of a product line:

> When combining software architecture and component-based software development, the result is the notion of software product lines.

According to Bosch's definition, introduction of an architectural approach limits the reuse of components to within a company if there is no architecture with industry-wide acceptance. Bosch expands his definition by comparing it to other types of strategic reuse. In his framework of software product line maturity (Bosch, 2002), presented in Chapter VIII, he highlights the difference between pure reuse, which he assigns to the *platform* state in the framework, and reuse with variation management, which he assigns to the *software product line* state. The need for variation management comes from reusing not only identical functionality, but also similar functionality, which in other words is the difference between using a platform and using an SPL according to Bosch.

In his definition, Krueger (2003) emphasizes the benefits of an SPL, and points out reuse, as well as variability:

> The objective of a software product line is to optimize software engineering effectiveness and efficiency by capitalizing on the commonality and managing the variation that exists within a product line of similar software systems.

Krueger's definition also points to the benefits of an SPL, when he mentions optimizing effectiveness and efficiency. The SEI definition (Clements & Northrop, 2001) further emphasizes the common marketing focus of the products, and the process by which the products are developed:

> A *software product line* is a set of software-intensive systems sharing a common, managed set of features that satisfy the

> specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.

For a product line to be an SPL according to this definition, the product line must also implicitly be developed according to SEI's framework, or possibly some other, so that the products of the product line are developed in a prescribed way. Even though the focus of these three definitions differs, they all agree on including the concepts of reuse and variability.

2.1 Components of SPLE

An SPL contains managed variation, variation that must be bound at some point in the production process. These variation-binding points, spanning from development time to runtime, form the basis of an SPLE classification scheme proposed by Krueger (2003). The dimensions of the taxonomy are:

- *Production*, describing how, when and by whom products are produced from reusable assets, how decisions to bind variability are made in production, how the reusable assets are represented, and how variability is represented in these assets.
- *SPL evolution*, i.e. how enhancements etc. are propagated to and from affected assets and products.
- *Scope*, described on a floating scale from proactive to reactive scope management, where the proactive approach tries to capture all foreseeable variability of future products, while the reactive approach only adapts the assets of the SPL when new variability is needed. These approaches are discussed further in Sections 5 and 6.
- *SPL initiation*, describing how the transition from standard software engineering to SPLE is managed, an issue revisited in Section 5.

Variation mechanisms in the *production* dimension have two aspects: level of abstraction and scale. The level of abstraction ranges from requirements variation linked to code generation, variation points supported by the Product Line Architecture (PLA), to code level variation where source files are overwritten or rewritten to create variants. The scale might range from single lines of code to subsystems, and depends on the level of abstraction. If design is the level of abstraction, with variation provided by an object-oriented framework, then the scale of variation is the concrete classes that extend the abstract classes of the framework. If code is the level of abstraction, with reuse supported by a Configuration Management (CM) tool, the scale of variation will be the files that the CM tool is managing.

In short, SPLE is about reuse and variation. Reuse is the main benefit of SPLE, and variation has to be managed in order to develop several different products from the same reusable assets.

3. Frameworks of SPL

This section gives an introduction to four attempts at creating frameworks that cover all aspects of SPLE, in order to support the introduction and operation of an SPL. The section also shows the two major camps in the field of SPL, a heavyweight approach advocating large initial investments in reusable assets, and a lightweight alternative, where investments are only made for the immediate future. The heavyweight approach is represented by the first three frameworks, especially SEI's, and the lightweight approach is represented by Krueger's BigLever (2005) framework.

3.1 SPL According to the SEI

SEI's Framework for Software Product Line Practice (SEI, 2005a) is based on the three essential activities: core asset development, product development, and management. The framework consists of 29 product line practice areas grouped in areas relating to software engineering, technical management and organizational management, shown in Table 7.

Most of these practice areas are present in standard software development. The framework therefore describes how each area is different when implementing and running a software product line; discusses the area in relation to the three

Table 7: SEI Practice Areas

Software Engineering Practice Areas	**Technical Management Practice Areas**	**Organizational Management Practice Areas**
Architecture Definition	Configuration Management	Building a Business Case
Architecture Evaluation	Data Collection, Metrics, and Tracking	Customer Interface Management
Component Development	Make/Buy/Mine/Commission Analysis	Developing an Acquisition Strategy
COTS Utilization	Process Definition	Funding
Mining Existing Assets	Scoping	Launching and Institutionalizing
Requirements Engineering	Technical Planning	Market Analysis
Software System Integration	Technical Risk Management	Operations
Testing	Tool Support	Organizational Planning
Understanding Relevant Domains		Organizational Risk Management
		Structuring the Organization
		Technology Forecasting
		Training

essential activities; and presents or gives references to specific practices for that area.

Requirements Engineering (RE) is an example of a Software Engineering practice area, where the differences for SPLE from standard RE focus on the division of requirements into those generic to all members of the product line, and those specific to each product. RE for SPL core asset development introduces a hierarchy of requirements. Practices for turning individual product requirements into generic requirements are therefore needed, and vice versa. This practice area is highly related to the scoping practice area, found under technical management. Technical management also includes processes, planning and tool support, while organizational management includes practice areas relating to e.g. business cases for SPL, marketing and organizational structure. The three groups of practice areas are meant to target different groups of roles in an organization.

To guide a company in managing these 29 practice areas, SEI also presents 12 Software Product Line Practice Patterns. The general definition of a pattern is a description of a problem, a solution, and the context in which the solution fits the problem (Gamma *et al.*, 1995). The practice patterns describe how to group and relate practice areas to accomplish a certain part of an SPL effort for an organization in a specific situation. The patterns were identified, as they were recurring solutions to problems from the case studies SEI has undertaken, and act as a way to reuse SPL experience.

Some of these patterns are educational and only provide an overview of the framework, e.g. a mapping of practice areas, practice area groups and essential activities. Others are of a more operational nature, describing which areas are involved when developing any internal asset, or when determining what products should be included in the product line. These operational patterns are then combined into composite patters that describe how to implement a product line and develop products from it.

Finally, SEI's framework also includes the Product Line Technical Probe, an instrument for assessing the SPLE readiness in an organization. The results of such an assessment include the strengths and weaknesses the organization exhibits with regards to product line engineering, along with recommendations on how to initiate or improve a product line effort.

In conclusion, SEI's approach to SPLE is architecture and process centric. It is heavy to implement, and also promotes *proactivity*, i.e. upfront investment in common assets to support predicted variability.

SEI's framework has been developed from several case studies performed by the SEI, with a strong bias towards defense industry. It is also based on a series of workshops on SPLE, which have evolved into an annual conference (SEI, 2005b).

3.2 SPL According to Bosch

Another prominent reference in SPLE is Bosch (2000). Bosch presents a similar approach to SEI's that is extracted from several case studies. Bosch's approach contains a process for introducing SPLE that includes developing a business case for SPLE, setting the scope for the product line, product planning, developing a PLA, and populating this architecture with components. The PLA is developed using Functionality-Based Architectural Design, starting from functional requirements. The architecture is then transformed to conform to quality attributes. This is different to SEI's quality driven approach to architecture (Bass *et al.*, 1998), which starts with quality attributes.

Bosch provides more technical details on how to develop components with complex interfaces, and how to provide variability on the component level. For example, Bosch suggests that object oriented frameworks can be used for complex interfaces, and that component variability mechanisms can include inheritance, configuration or templates. Variation interfaces, in the form of documentation or operations performed at instantiation, provide standardization and access to these variability mechanisms. If upfront investment has not been made in variability for a component in a particular setting, the component instead needs to be adapted. This can be done by copy-paste, inheritance or wrapping in conventional programming languages, but in general adaptation should be transparent, black-box, composable, reusable and configurable.

3.3 SPL According to Fraunhofer IESE

The empirical software engineering research branch of the German Fraunhofer institute, Fraunhofer IESE, also has a project in SPLE, which has resulted in the PuLSE (Product Line Software Engineering) method for introducing SPLE (Bayer *et al.*, 1999). The method consists of six components. The *Baselining and Customization* component focuses on technology transfer. It enables companies to tailor the method to their specific conditions. Four of the components relate to technical activities: *Economic Scoping*, *Customizable Domain Analysis*, *Domain Specific Software Architecture* and *Instantiation*. These components focus on the early activities of SPLE, and indicate that the method is best suited for proactive SPLE. *Evolution and Management* is a management support component used to maintain and evolve an existing SPL. Although the method seems to favor proactive SPLE, it can be introduced incrementally onto an existing software process.

Fraunhofer IESE also gives an overview of eleven code level variability mechanisms (Anastasopoulos & Gacek, 2001), which extends Bosch's overview. Their mechanisms include parameterization, conditional compilation, and object oriented mechanisms such as aggregation and inheritance. In their overview they compare the mechanisms to illustrate the variation for which they can be used, when they can be used, and which programming languages support each mechanism.

In summary, for these three heavyweight SPL method, the SEI framework is the more elaborate when it comes to managerial support. It is also the framework that depends most upon a proactive approach, with heavy initial investments in reusable assets. The frameworks by Bosch and Fraunhofer IESE provide more technical detail on e.g. architectural design and mechanisms to achieve variability.

3.4 Lightweight SPL

The methods mentioned above require heavy investment before development of products from the product line. The investments are in processes and procedures, but foremost in a PLA with reusable components. In order to lower this adoption barrier, Krueger advocates extractive adoption of SPLE. This is achieved by reusing one or more existing software products as an SPL baseline, with limited reengineering. This technology is packaged in a tool, BigLever Software GEARS (BigLever, 2005; Krueger, 2002a), which sits on top of any CM tool, and therefore works at the level of individual files. Variation in a product line is modeled in *feature declarations*, products that can be developed or generated from the product line are described in *product definitions*, and *automata* are actuated to configure products from source files, declarations and definitions. The tool can provide the infrastructure for the *proactive* approach to SPLE described in Section 5. Its main benefit is however that it enables the *reactive* approach, where an SPL is incrementally extended to allow for more variability as it is needed, and the *extractive* approach, where a proper SPL is extracted from a set of existing products with commonalities. The extractive approach is seen as a starting point, which is to be followed by inclusions of either other products extractively, or new variability reactively. These two approaches therefore allow for investments in reusable assets to be made when this reuse is needed, rather than in advance, as is the case with the proactive approach.

Krueger acknowledges that the proactive approach can be applied when the scope of variability can be determined far in advance, and when an organization can afford to spend resources up-front, possibly by delaying current products while setting the scope and architecture of the product line. The benefits of the proactive approach are then achieved as inclusion of new products should not require modeling of new variability.

Although Krueger's approach uses code-level variation, all approaches rely on the PLA as the most important asset to manage reuse and variability. In search of even lower barriers of adoption, Staples & Hill (2004) present a case study of a company which did not rely on architecture-level variation points when introducing an SPL. The company introduced SPLE *extractively*; reuse was managed by the CM tool, and variation was managed through build scripts. The approach is based on providing variation by overwriting source files from a core CM branch. It has weaknesses in that the level of reuse decays as changes useful to all products and customers are made to customized assets, and is therefore only used to introduce SPLE.

4. Comparison of Ways to Introduce SPL

Two scenarios show the different needs companies might have when initiating a product line:

- A company develops one product, but then has to customize it for each customer in a contract- and project-driven fashion. This results in a diverging set of products, but products that will have the same basic functionality. Changes to one product, that could benefit others, are not easy to transfer. The company therefore wants to increase software engineering efficiency and effectiveness by only correcting errors found in all products once, and by only introducing new features used by all products once. The company still has to manage variations between the individual products. The situation is depicted in Figure 3.

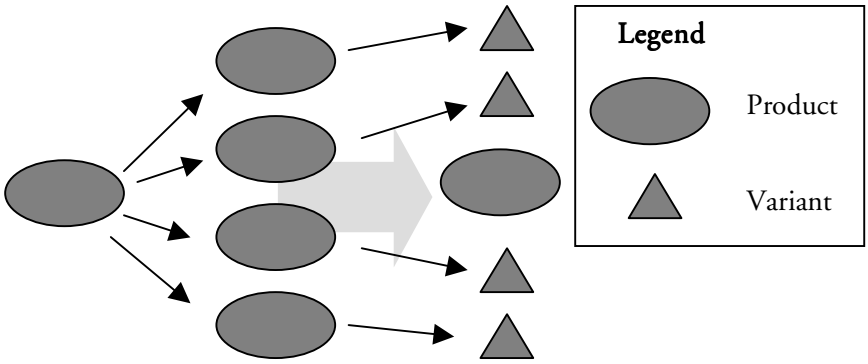


Figure 3. Turning diverging products back into one product with variants

- A company develops an established product for a mass market. The market segment is broad, but borders to closely related niche segments. The company therefore wants to capitalize on differentiation, which e.g. can be accomplished by internationalization to capture other geographic market segments, or reaching other demographics by differentiating on cost. From the basis of one product, the company wants to introduce variation that can be planned ahead, while still seeing the future product line as one product. The situation is depicted in Figure 4.

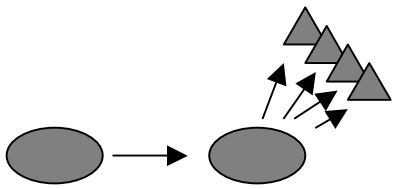


Figure 4. Differentiating an existing product by introducing variability

These two example contexts are quite different, but the end state is similar – to maintain and evolve *one* product line, rather than several products. To manage the differences in the needs of companies about to introduce SPLE, a number of introduction models have been presented in literature. Bosch (2002) divides these according to two dimensions – *evolutionary* vs. *revolutionary*, and *existing set of products* vs. *new product line* – where the four combinations are:

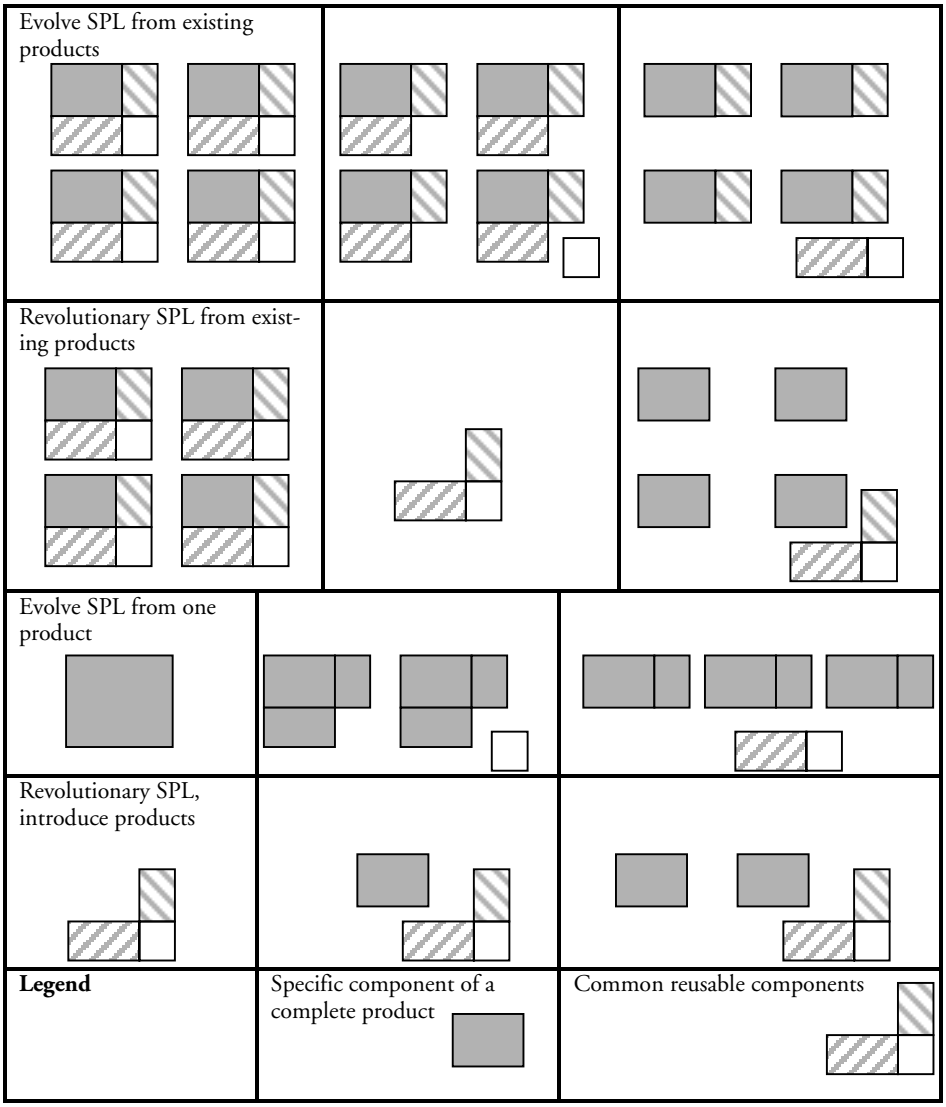
- Evolve existing set of products into product line (*evolutionary, existing*): A PLA is developed based on the architectures of several existing products, and one component at a time is generalized, or reengineered, from the existing customized components.
- Replace existing set of products with product line (*revolutionary, existing*): All product development is halted, a PLA is developed, and products and components are reengineered according to present and predicted requirements on product line members.
- Evolve new software product line (*evolutionary, new*): The product line is initiated by one or a few products, and commonalities and variation are only modeled for these products, but later evolved as new products enter the product line. The requirements on these new products might force reengineering of the PLA.
- Develop new software product line (*revolutionary, new*): Assets are developed with all expected product line members in mind. The aim is to avoid reengineering, but the risks are high since domain knowledge has not been established.

Table 8 shows three stages of development during introduction of SPLE on a line of four products according to these four approaches. The grey top-left component of each product is specific to each of the four products, while the other three components are reusable throughout the line of products.

A company in a situation such as shown in Figure 3 would have to choose between the two former alternatives, with an existing set of products. Companies described by Figure 4 would choose from the two latter. These four strategies focus on reusing common assets, rather than managing variation between products. The SEI refers to these four strategies, but promotes the *revolutionary*, or *proactive* approach:

- *Proactive*: Similar to Bosch's two revolutionary types of initiation, where a scope is set for current and future members of the product line, a scope which defines if any requested product should be developed by the organization as part of the product line. Apart from establishing the scope, this type of initiation requires developing a business case for the product line, common requirements and a common PLA for the members of the product line, along with components common to all members. Development of these common assets is supported by the SPL Practice Pattern Product Parts, with its variants Green Field, Barren Field and Plowed Field, where

Table 8: Bosch's introduction of SPL



existing assets can be mined to various degrees (Clements & Northrop, 2001).

Krueger on the other hand is more averse to designing solutions to requirements that have not yet been asked for by customers, and he presents one way of introducing SPLE according to this philosophy:

- *Extractive*: Assets are mined from one or several existing products. If the set of existing products is large, commonalities and variation can be extracted incrementally, product-by-product, as opposed to Bosch's component-by-

component alternative. The strategy has the same aim as Bosch's two evolutionary initiation approaches, with low risk and low barriers of adoption. It is solely a product line initiation strategy, compared to the proactive and reactive approaches, which are used to evolve an existing PLA to allow for new variation. The previously mentioned case study of Staples & Hill (2004) is also an example of an extractive method, a method where a PLA was not used to implement variability.

The adoption models can be seen in light of the Technology Adoption Life Cycle (Moore, 1991), where early adopters are ready to put resources into being first with SPL, and thereby gaining a competitive advantage. However, most companies will hesitate until the technology has been packaged in a more user-friendly fashion, i.e. in well-established tools and consultancy services, rather than in singular case studies. This concern is in line with concerns about the total cost of introducing SPL: the tools and knowledge need to be affordable; but foremost, the introduction must not stall product development for an extended period of time, or consume extraordinary resources. In Krueger's words, as with any technology, "for the mainstream software engineering community … the adoption barrier must be much lower than that experienced by the early pioneers" (Krueger, 2002b).

5. Comparisons of When and How to Introduce Support for Variation

The previous section focused on initiation of a product line, which is closely related to how support for new variation can be introduced in an established product line. The two opposing camps are, as before, the *proactive*, with SEI in the forefront (Clements, 2002), and the *reactive*, led by Krueger (2002b). In the proactive approach up-front investments are made in roadmaps, PLAs, and component variability, to cover all products in the scope of the product line. In the reactive approach risks are minimized by only satisfying existing customers. This is achieved by implementing functionality they seek, and making the required changes to the PLA to support reuse and variability as such needs become apparent. The proactive approach is linked to internally elicited market requirements, while the reactive is linked to requirements from explicit customers. In other words, the proactive approach is more market driven than the reactive, contract driven approach.

According to the proactive approach, changes are made before they are needed, while reactive changes are done when they are needed. In the case study presented by Staples & Hill, the initial code level variation mechanism soon led to reuse decay, where changes to source files in individual products reduced reuse benefits. When this type of reuse decay was discovered, architectural support for that specific variation was enabled by performing *retroactive* changes, i.e. after variation was needed. In this way the company is slowly moving from a

SPL based on code-level variation, to one based on a full-fledged PLA, while still receiving some of the benefits of an SPL in the transition.

Allowing more variation or exploiting more commonalities for reuse requires changes to the core assets and also, in the general case, to the PLA. One method of performing architectural change is *refactoring* (Opdyke & Johnson, 1993), where the architecture is changed without changing functionality, and the system then re-tested before changes to functionality are allowed, changes that will then be implemented on a stable base of reusable assets. Refactoring describes *how* to perform architectural changes, and comes from the Agile community, which generally favors the reactive approach to deciding *when* to perform architectural changes. Refactoring can however also be applied when changing the architecture proactively or retroactively.

Contrary to Krueger's attempts to limit the organizational effects of introducing SPLE, Chapter VII presents a process for architectural change that instead acknowledges the impact on organization and process. Bosch (2005) also recognizes organizational impact, when he presents an integrated framework for staged adoption of software product families. The framework is centered on five decision dimensions: feature selection, architecture harmonization, shared component scoping, organization and funding. Along these dimensions Bosch suggests procedures to be put in place as the company gains SPLE maturity in three stages, from *initial adoption*, over *expanding scope*, to *increasing maturity*. The assumption behind this framework is that initiation of SPLE is best done according to light-weight principles, but that full institutionalization is later achieved with process rigor and organizational bureaucracy.

V The Business and Economics of Software

Software products are what drive software businesses, but good software business is not only about utilizing the most innovative products. Engineering is about developing technical solutions, but in an effective and efficient manner, i.e. finding a balance between innovation, quality and cost. However, the business of software also concerns business and marketing strategies – releasing and marketing the right product at the right time. The first three sections of this chapter focuses on business and marketing strategies related to high tech markets, specifically software markets, and the impact such strategies have on strategies for software architecture and development. The sections towards the end of this chapter instead focus on economics tools for use in not only strategic but also operational, day-to-day software engineering.

1. General Marketing Models for High-Tech Markets

A number of models exist for high tech market strategy (D'Cruz & Ports, 2003). This section gives an overview, while Sections 2 and 3 give in-depth information on two models that are tightly linked to strategies for architecture and development.

1.1 Product Portfolio Matrix

Boston Consulting Group's Product Portfolio Matrix (Stern & Stalk, 1998) is a tool to analyze the products in the portfolio of a company. Analysis is done along the two dimensions *market growth* and *relative market share*. Startups usually only have one or a few products with a small market share on a rapidly growing market. Their goal should be to gain market share with these products, while established companies should have a good spread of high-market-share products on both stable and increasing markets, and new, low-market-share products on expanding markets. Any company should divest in products with low market share on markets that are not expanding. The model is simple, and

does not account for the revenues made from the products, but only focuses on absolute market share. Products with high absolute market share are assumed to generate revenues, and gaining market share is assumed to require investments in marketing or technology. The most direct impact of portfolio management to software development is in prioritizing resources (Vähäniitty, 2004).

1.2 The Whole Product

The concept of the Whole Product (Levitt, 1986; McKenna, 1991) is also used in engineering and marketing of high-tech products. The model consists of four layers of a product. A whole product is not only the underlying technology, the *generic product*, which is the engineering view of the product. The *expected product* is the least a customer would be interested in buying. This is often determined in view of competitors' offers. The *augmented product* includes features a customer would not expect, features which can be used to beat competitors. The *potential product* includes all factors concerning the customer, such as total cost of ownership and services which help customers fully utilize the product.

1.3 Disruptive Technology

The theory of Disruptive Technology (Christensen, 1997) says that a new, *disruptive*, technology will attract customers on behalf of proven, *sustaining*, technology as soon as it reaches the customers' minimum quality requirements – even though products based on the old technology still outperform these new products. In Whole Product terms, when a disruptive technology evolves from *generic* to *expected* products, it may gain market share on behalf of sustaining technology packaged as *augmented* or even *potential* products. This speaks in favor of startup companies, because established companies with proven solutions are less motivated to replace technology that works and is profitable. The problem with such disruptive technology for established companies is that it provides new applications of technology, applications where business models and business value is hard to foresee. The performance of the new products is also lower than that of the existing technology, and is therefore not viewed as a threat. The performance and quality of disruptive technology can however improve rapidly, as development and manufacturing processes are tuned.

A good balance in the product portfolio matrix of an established company can alleviate the risk of being overtaken by startups with disruptive technology. Products with disruptive technology from this portfolio should initially be separated from the rest of the organization and focused on niche markets, to not interfere with and cannibalize on profitable, sustaining, technology (Lindsay & Dennis, 2001).

2. The Technology Adoption Life Cycle

The theory behind the Technology Adoption Life Cycle assumes that consumers behave differently regarding when they are willing to adopt a new technology. This means that different marketing strategies will be appropriate as the target market changes. The technology adoption life cycle model exploits the three models presented in the previous section. The model gives guidelines on how to act in order to move products to more favorable sectors of the product portfolio matrix, and uses the concepts from the whole product to determine what focus a company should have in the various stages of the life cycle. It also explains the dynamics of a market as disruptive technology becomes established and gains market share. Figure 5 shows the segments of adopters and the phases of the technology adoption life cycle, as described below.

2.1 Types of Adopters

According to the technology adoption life cycle, people behave differently when faced with new technology (Rogers, 2003). These behavioral differences are divided in the segments *innovators*, *early adopters*, *early majority*, *late majority* and *laggards*, see Figure 5. Innovators are those who acquire new technology for the technology itself. Early adopters are not so much interested in the technol-

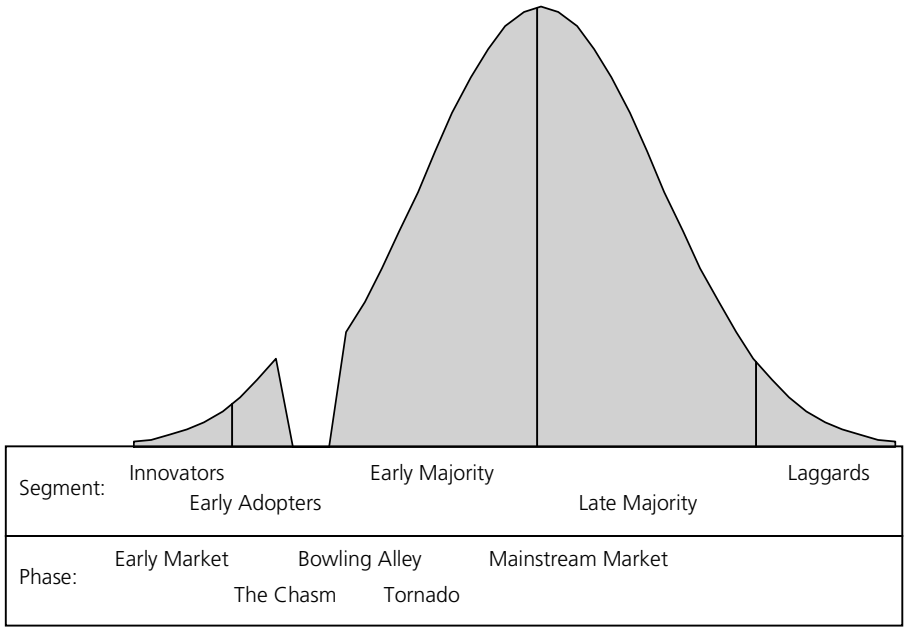


Figure 5. Segments and phases in the Technology Adoption Life Cycle

ogy, but in how it can be put to use, and are willing to accept cumbersome solutions with initial defects – where innovators *enjoy* cumbersome solutions with initial defects. The early majority sees technology from a practical point of view, and wants references before buying in to new inventions. The late majority wants simple solutions as they are uncomfortable with new technology, and laggards don't want anything to do with new technology. The early and late majority constitutes the bulk of a future market – the early majority is pragmatic, while the late majority is conservative. Figure 5 can be compared to Figure 1 in Chapter I, where the figure here – showing adoption – can be viewed as the cause of the product life cycle in Figure 1, Chapter I.

Each segment has their own incentives to buy; therefore the shifts between target groups mean not only quantitative differences in strategies for marketing, but also qualitative, in that each group have their own requirements on a product or offering.

2.2 Phases of the Life Cycle

The general idea is to establish the product with each of the groups of customers in sequence, and then exploit the present group of customers, as reference or for setting standards, in order to reach the next group of potential customers. The problem is that the transitions between market segments are not smooth, primarily because the target segments have quite different goals.

Customers found among innovators and early adopters in the early market can drive up expectations on the profitability of the product. The main obstacle is to evolve a business from this early market onto the mainstream market of early and late majority adopters. The technology is not yet mature enough for the majority of customers to readily accept it, and the majority also wants one new infrastructure to set the standard for coming technology. This gap between early market and mainstream is called *the chasm* (Moore, 1991). The early majority is looking for non-disruptive technology that improves their business, and they choose such technology by referencing others in their situation. The customers of the early market however chose the new technology as it is disruptive, and can give enormous benefits at high risks. Such customers are therefore not good references to enter the mainstream market.

The strategy suggested by Moore (1991) to cross this chasm is to focus on one niche market at a time, and becoming the market leader in each of these niches. With this focus, the technology can be packaged as an *expected product* for each of these niches. It is important to select the right initial niche, although there will be very little marketing data to back up this decision. In the early market no potential customer can be turned down, but the focus required when crossing the chasm will require a company to change strategy and say no to customers that do not fit the target. The following niches should be selected close enough to the initial niche, in order to maintain focus and exploit experiences made in previous niches. This phase is called *the bowling alley*. The strategy should give initial revenues needed to develop a sustaining technology, and pro-

vides references for similar niches, and finally the entire market of early majority customers.

Eventually the technology might enter *the tornado* (Moore, 1995), where the entire early majority changes to the new technology. They do it as fast as possible, to minimize the disruption, and they go for one vendor of the new infrastructure, to get common references and a de facto standard. There will therefore be only one major winner in this phase, and the strategy should be to care less about the customer and more about beating the competition and being able to ship as many units as possible. In this phase customers will select the winning product even if it doesn't match their specific requirements, because they know they will have to change infrastructure, and that the winning vendor will be able to provide future support for the selected technology. The strategy is therefore quite different from the previous phase, where customer focus was key to winning niche markets. During this tornado, price will become important. Earlier markets can be priced on the value provided to the customer, but during the tornado competition will be driving prices down and commoditizing the product.

Eventually the hypergrowth of the tornado stops, as the market saturates. During this established market it is time to capitalize on the market share gained during the tornado. Prices can generally not be increased, so in this phase costs are important to control, through e.g. process discipline. Market shares can still be gained by adding value to the product with extra features and product differentiation in order to again reach niche markets.

2.3 Shifting Focus in the Life Cycle

The theory has direct impact on architectural strategies to reuse. On early markets it is all about selling to the customer you have. They will buy the technology even if it is not packaged in a user friendly product and targeted directly towards them, because they have the technological know-how to help you integrate the offering into their organization. Every new customer however means a new commitment to integrate the unpackaged technology into a new setting, and the emerging products might very well be diverging.

In order to cross the chasm to the early majority, the technology must be packaged in a whole product. It should also be targeted at one specific niche in the future market. The technology therefore has to go from several customized solutions to one product standardized for a small market niche. This resembles the scenario in Figure 3 of Chapter IV, where an initial product has diverged into several customized offerings, and then has to be brought back to one standardized product, with possible variations to fulfill previous commitments.

The whole market should thereafter be conquered by expanding into more and more niche markets, which again leads to several specialized products that however are standardized for their specific markets. The technology itself is important, but also knowledge about customers on these niche markets. As the tornado hits, a new cycle begins, with focus on one standardized product. If this

product survives and becomes one of the winners of this rapid market expansion, it can later be diversified to once again gain market niches, as in the scenario depicted in Figure 4 of Chapter IV.

The theory suggests several types of actors on a market, especially during the tornado and after. The tornado generally generates one main winner, but there will not be a market if there is no competition to refer to. The initial technology might come from startups that do not have the financial power to provide an entire market with a whole product solution. Some actors will therefore employ vertical strategies, where a whole product is supplied to a niche market. Others might see opportunity in horizontal strategies, where they provide many such niche markets with an underlying infrastructure. During the tornado and after, smaller actors will attack the winning product by providing e.g. low cost alternatives. Others might develop supplementary products and services to the main product.

The theory of the technology adoption life cycle is however not undisputed. It concerns adoption, which shall lead to market share, but it is unclear if crossing the chasm into a mainstream market will be profitable for every high-tech product. Also, contrary to Moore's view that totally different strategies are needed for the early majority, von Hippel (2005) suggest that the innovators and early adopters can very well be used to find inspiration for more innovation.

3. Products or Services

Cusumano (2004) has been analyzing the evolution of software companies by investigating changes in the balance of sources of revenues. This has added a further dimension to strategies for software business. Cusumano bases strategy assessments on five basic questions: are we marketing products or services; are we targeting individuals or enterprises, or mass or niche markets; are we targeting a horizontal or vertical market; how do we intend to generate recurring revenue streams; how do we plan to reach mainstream customers, or do we intend to avoid the chasm altogether; do we intend to be a leader, follower or complementor on the market; and what is the character of our company. The main dimension in this strategy framework is whether a company supplies products or services to its market.

Developing and marketing a software product can give the highest returns on investments, because after development, production is virtually for free. It is therefore the dream of most software companies. The risks are however significant, in that revenues can be very unpredictable, and cease altogether in bad times, when customers don't upgrade to the latest version of your product. A prerequisite is also that all customers want the same product, and not versions customized to their particulars. Cusumano's data show that software companies that are thought to be purely product companies therefore tend to become more and more service suppliers, as this gives recurring revenue streams from established customers. Hybrid solutions, with a mix of product and service of-

ferings, are a common way to gain the large revenues of products, and the recurring revenues of services.

The whole product can therefore be accomplished by either products focused strategies, or services focused strategies. The technology itself, the generic product, can be packaged and wrapped with the features necessary to make it directly useful to the customers as a product. In order to evolve this expected product into an augmented product and further, methods for mass customization, such as the software product line strategies of Chapter IV, or some of the architectural and development strategies of Chapter VIII, can be employed. Another alternative for turning a generic product into a whole product is to provide this packaging as a service to each specific customer, as Company C currently does when delivering their platform. Such services are often provided by 3rd party consultants acting as integrators, as shown in Chapter IX.

The strategic models presented in this chapter so far have had impact on the strategies available for architecture and development in the companies studied in this thesis. These are most visible in Chapters VIII and IX. The following sections give more fine-grained tools for decision support concerning the economics of software development. These are related to the aspects of engineering which demands us to develop solutions at the right cost. Innovative solutions coming from the engineering or development department of a company must also be able to show benefit, and risks must be under control in software development.

4. Cost

Cost modeling has come far in the field of software engineering (Halling *et al.*, 2004). Software development project success is often defined in terms of meeting budget and deadline. This view can suffice if business can predict the value of developed products, and if products are developed in individual projects. COCOMO (Boehm, 1981) is one such model, based on historical industry-wide data, to estimate the cost of projects, based on the expected functionality which should be developed by the project, and the project's inherent risks in terms of application type. This, and similar models (Fenton & Pfleeger, 1998) gives opportunity to balance total project costs against project duration and staff count.

One problem with such cost models is to estimate the volume of functionality which is to be developed in a project. In most models this is relative to lines of code, but there are metrics which are telling more about actual functionality, such as function points (Albrecht, 1979; Caine & Banks Pidduck, 2004). Traditional project planning uses e.g. work breakdown structures, but recent software engineering methods use object counts or use cases (Jamieson *et al.*, 2004).

It is also difficult to determine the actual benefit generated at any one time during a project, although the cost might be known at any such time. Furthermore, the requirements stated at the initiation of any project may not be true, as they may not be known by the customer in advance, or may change during

the project (Noppen & Aksit, 2004). Most costs can be transferred into monetary units, but this should be done late during cost estimation, to retain multi-dimensional information about different types of costs as far as possible (Poladian *et al.*, 2003). Cost factors are also not only limited to development, but include future use and maintenance (Asundi *et al.*, 2000). Cost and value are therefore linked, and sometimes hard to separate, as one type of value for a customer is the total cost of ownership for a product (Ferrin & Plank, 2002).

5. Value

Costs of development are important to know, control and estimate. But the value of what is developed during a software project is what any customer is willing to pay for – what should govern the pricing of software products and services (Cooper, 2000). When assets are developed for internal reuse, the costs of these are also not enough to assess whether they are of benefit to projects developing products from them – the value of each such component must be determined.

Value is more difficult to assess than cost. The true value of a complete product can be determined by judging its success on the market. This in turn means that the value of a project often can be determined – if it concerns developing a new product, or adding value to an existing product whose value can be determined. The value of a specific architectural strategy or architectural change is much more difficult to establish. Individual features of a product can be valued through e.g. AHP (Saaty, 1980), and the same can be done for quality attributes – although the assessment becomes more abstract. To concretize abstract attributes such as quality in general, or reusability, maintainability, and extensibility, scenario-based techniques have been employed heavily in software architecture research. Certain architectural strategies are said to have positive or negative impact on certain quality attributes. Concrete change or usage scenarios show aspects of certain quality attributes, and when the scenarios are valued against each other, the qualities are indirectly valued. If we assume that software architecture is driven by, and enables, quality attributes, and if we know which architectural changes have impact on which quality attributes, then we have a link from the concrete scenarios a user or customer requests, and the architectural strategies which help fulfill these scenarios. SAAM (Kazman *et al.*, 1994) is one scenario based architectural assessment method which assesses the relative value of modifiability in architectural decisions, while ATAM (Kazman *et al.*, 1998) can be used to value several quality attributes of an architecture. Scenario-based methods are further discussed in Chapter VI.

This indirect valuation often leads to a relative value, where different architectures can be compared to each other. The problem is when an architecture already exists, and current projects still can deliver value in the products they develop, but the productivity could possibly be improved through investment in architectural assets. If the value generated by such architectural improvements cannot be compared to the value generated by letting development pro-

jects do business-as-usual, then it is difficult to get resources and funding for long-term architectural efforts. The Cost Benefit Analysis method (CBAM; Moore *et al.*, 2003) is an extension of ATAM, where the relative value of different architectural strategies is assessed in the context of a particular set of sought-for scenarios and scenario outcomes. It still provides a relative measure, and assumes the existence of a fixed architecture budget, but architectural investments could be weighed against user features in this method.

For valuation in general, Reifer (2004) presents a framework of valuation model sophistication, based on Pitkethly (1997) and used in valuation of software trade secrets:

1. Cost approach, based on the cost to replace an asset with a comparable asset.
2. Income approach, value in terms of future cash flows.
3. Market approach, valuation based on recent sales of similar assets.
4. Time value of money, with discounted cash flows.
5. Uncertainty, valuing discounted cash flows for risk.
6. Flexibility, valuing discounted cash flow and decision tree analysis.
7. Changing risk, valuation with real options theory to adapt for changing market and economic conditions.

The framework shows available types of methods, and shows that risk must be considered to set a proper value on products or assets that are not yet developed.

6. Risk

The value of a product under development might change dramatically, if e.g. the development project misses its deadline, or if there is a recession in the economy when the product is released onto the market. These risks must therefore be assessed too. One simplistic way of accounting for risk is to evaluate based on discounted cash flows, such as net present value analysis (Birrer & Carrica, 1990), and use an increased discount rate (Padberg & Müller, 2004). Such methods are appropriate for short term investments and decision support for ongoing business (Bahsoon & Emmerich, 2003). A proper risk assessment is however done with probability distributions, and real options theory is one way of modeling such value (Asundi & Kazman, 2001).

Traditionally investment is done when the net present value is positive. But such decisional support assumes that investment decisions can be reversed. Investments in software development must however usually be treated as sunken costs if it does not result in a marketable product or reusable asset. Real options theory can then be used to determine whether it is preferable to delay investment. Real options theory is derived from financial options theory (Black & Scholes, 1973). An option is an asset that provides its owner the right, without a symmetric obligation, to make an investment decision under given terms, for

a period of time into the future, ending with an expiration date. The owner of the option can exercise the option by investing the defined strike price, if this is financially favorable. The option can be exercised at or before the expiration date, depending on the rules of the option. Black & Scholes (1973) present a stochastic model to construct and evaluate financial options. Bahsoon & Emmerich (2003) have from this derived a method to evaluate real options – options on non-financial assets – specifically for the options provided by architectural investments. The method views the value of an architecture as the set of requirements it supports, and a set of representative requirements – or future scenarios – must be elicited.

Although real options theory looks promising, and has been used to evaluate software trade secrets (Reifer, 2004), the problems are that parameters such as discount rates of software projects, dividends payable by projects, and correlations between architectural investments must be determined. This does not lie in the scope of expertise of software engineers (Asundi & Kazman, 2001).

These risk assessment techniques aim at determining the flexibility of a software architecture. This flexibility can be viewed in different ways. The architecture can be flexible in that it is easy to change and adapt when new requirements arrive, or it can be flexible in that it does not need to change. Flexibility is usually retained by opting out alternatives late in the development process. This is done by Toyota when developing and designing new models of cars. Several designs are allowed to compete with each other well into a development project, rather than selecting one design as early as possible (Ward *et al.*, 1995). The strategy is resource intensive, as several design groups work in parallel, whereof most of them will be scrapped in the course of a project. With real options theory, the strategy has however shown to be efficient and effective (Ford & Sobek, 2005). The gain is attributed to less needed redesign during the project, as more options are available; and better quality of the resulting design, as more design improvements are given opportunity to emerge. Such strategies should be possible to transfer to design of software systems, if the differences in balance between cost of design and cost of production in car manufacturing and software development are properly considered.

People treat risks differently when comparing the risks of changing the current state of affairs to not doing so. Even though change is needed, people are biased towards not committing to action – omission bias (Suarez & Patt, 2004). If the risk of taking an action is similar to the risk of doing nothing, people tend to prefer inaction. This is however also true if the action involves risks, but risks that are lesser than the risks of inaction. One explanation could be that action carries responsibilities, more clearly so than inaction. A similar effect is people's bias to status quo (Suarez & Patt, 2004). If we retain our flexibility to make future decisions, maintaining status quo can be a rational choice. Such flexibility can be assessed by real option theory. But the bias towards the status quo is stronger than that, as people want to delay action in order to gain more information on alternatives, and want to diffuse responsibility for such decisions. Change initiators must therefore create a sense of urgency (Kotter, 1996) also

when rational evidence points to the benefits of taking action to reduce risks or gain other benefits. The following chapter presents a method for assessing flexibility and risk related to the development process, and Chapter VII investigates the problems facing an architect when initiating architectural change.

VI Process Flexibility and the Linkage Between Process, Organization, and Architecture

Flexibility is a desired quality of software processes. Process flexibility implies a capability to adapt to new contexts. Another aspect of flexibility is the cost of maintaining process effectiveness as new situations arise. A lack of preparedness for future events may constitute a high risk to a software development organization. This study presents a method for assessing the flexibility of an organization and its processes. The assessment method is scenario-based and provides an estimate of process flexibility in terms of risk. The method is evaluated in a case study, where the process flexibility at a telecommunication software developer has been assessed. The proposed method was able to identify a number of relevant areas to be improved in order to reduce risks of inflexibility for the particular process. The study provides insights into challenges regarding scenario-based methods. When compared to the line-oriented development used at Company B, the project-based case also shows a relation between software processes, architectures and organizations.

From Nedstam et al., "A Case Study on Scenario-Based Process Flexibility Assessment for Risk Reduction", Proceedings of the 3rd Intl. Conf. on Product-Focused Process Improvement, Kaiserslautern, Sep 2001, extended with information from Höst et al., "Exploring Bottlenecks in Market-Driven Requirements Management Processes with Discrete Event Simulation", The Journal of Systems and Software 59, pp 323-332, 2001.

1. Introduction

Software processes are intended to be used in many projects under a number of various circumstances. In many cases, experience and knowledge can be trans-

ferred from project to project in terms of common models of the process (Basili *et al.*, 1994). This would not be possible if the process was not reusable and adaptable. Zahran (1998) classifies process adaptability into e.g. suitability to support varying sizes of projects, suitability for different types of products, and flexibility to accommodate a variety of methods, techniques and tools. Process maintainability, which is related to adaptability, is discussed by Sommerville (2001) in terms of the process' ability to evolve in order to reflect changing organizational requirements or identified process improvement proposals. A number of similar definitions of flexibility in general and process flexibility in particular are discussed in Nelson *et al.* (1997).

This study focuses on the flexibility of the process in terms of its ability to maintain support for software development when changes in the environment occur. The changes may or may not lead to a need for process adaptation. Examples of changes are that the organization enters a new market, emergence of new technology, or changes in the employment market for the organization.

There are similarities between the usage of processes and the usage of software architectures. An architecture may be used in a number of projects, and it may be adapted for different situations. It also represents experience and knowledge that is applicable in a series of projects. In this chapter an assessment method, SAAM (Bass *et al.*, 1998), which is used for assessing architectures, is used as a basis for an assessment method for software processes. The derived method is evaluated in a case study in an industrial setting.

Standard process assessment methods such as CMM (Paulk *et al.*, 1993) and ISO 9000 (ISO-9001, 1994) focus on the predictability and quality of the outcomes of a process, and strive for process maturity in a general sense. The purpose of the proposed assessment method is to determine the flexibility of a software organization and its process. The flexibility may be measured in terms of the cost of maintaining the same degree of support when the process is used in a changed setting, or when it is adapted to fit a changed setting. This cost, combined with the probability of experiencing this changed setting, can be interpreted as a risk, and the assessment method proposes areas to focus on in order to reduce process-related risks.

The use of a scenario-based approach has shed light on what types of conclusions can be drawn from such methods, and the challenges involved in generating and interpreting scenarios. The case itself, which shows a project-driven organization, has been contrasted to the line-oriented case of Company B, which has been presented in Chapter III. This has provided insights into how power can be distributed in software organizations, and how this affects efforts on quality and architecture.

The structure of this chapter follows. Section 2 discusses related work and the theory the presented method is based on. Section 3 describes the proposed assessment method, built from the theory discussed in Section 2 and a case study. That case study, performed at Company A, is presented in Section 4. Section 5 discusses the conclusions from the case study, and issues of further research, while Section 6 includes a discussion of scenario-based methods in

general, and also general conclusions on differences between line- and project-oriented development.

2. Background and Related Work

The basic ideas of the proposed method are based on the Software Architecture Analysis Method (SAAM) (Bass *et al.*, 1998). The main contribution from SAAM is the usage of scenarios and a framework for developing and analyzing these scenarios. The scenario approach was selected as scenarios have been proven useful as tools for change management and decision support (Jarke & Kurki-Suonio, 1998). Nelson *et al.* (1997) provide some examples of how flexibility can be divided in terms of structural and process flexibility. This work has been a basis for breaking down process flexibility in terms of organizational factors, a set of factors used throughout this article.

SAAM primarily investigates the modifiability of a software architecture, an attribute closely related to flexibility. SAAM is performed in the following way:

1. **Develop Scenarios**: A list of expected uses or changes to the analyzed architecture is produced. It is important to consider all stakeholders affected by the architecture. The scenarios are also organized to indicate relations among them.
2. **Describe Candidate Architecture**: The architecture is described so it can be analyzed. This step is done in parallel with the previous, as need for detailed architectural information emerges when new scenarios are found.
3. **Classify Scenarios**: The scenarios are classified as direct or indirect with respect to the architecture. A direct scenario is supported by the architecture without alterations. An indirect scenario requires some changes to the architecture in order to be supported.
4. **Perform Scenario Evaluation**: In this step every indirect scenario is analyzed to find out the effort or cost needed to support it.
5. **Reveal Scenario Interaction**: If many scenarios require changes to the same component, that component probably has a high structural complexity. Such scenarios are said to interact.
6. **Overall Evaluation**: The cost of using the architecture is calculated by assessing the cost for each scenario and scenario interaction.

The SAAM has since been refined into the ATAM (Clements *et al.*, 2002), where also the tradeoffs made in the architecture are identified, and the sensitivity of the architecture is assessed based on a prioritized set of quality attributes and scenarios. This evolution could show how the method presented in the following section could be further enhanced. To translate the principles of SAAM from architecture modifiability to process flexibility, practical guidelines to evaluation (Scriven, 1991; Robson, 2002) has been used, following the example of Ares *et al.* (2000).

3. The Assessment Method

This section describes the assessment method, as it was used in the case study that is presented in the following section. An overview of the method is given in Figure 6. The method consists of five activities, which are approximately carried out in sequence. First the assessment is planned. After the method's planning phase, a description of the assessed process is generated. This description matures throughout the whole assessment, and is an aid for selecting relevant interviewees. The assessor then develops scenarios from a list of issues that emerge during interviews. After an initial categorization of these scenarios the participants assess the probability and cost of each scenario, from which a risk can be calculated. The scenarios with the highest risks are analyzed to provide decision support and recommendations for process improvement. These five steps are described below.

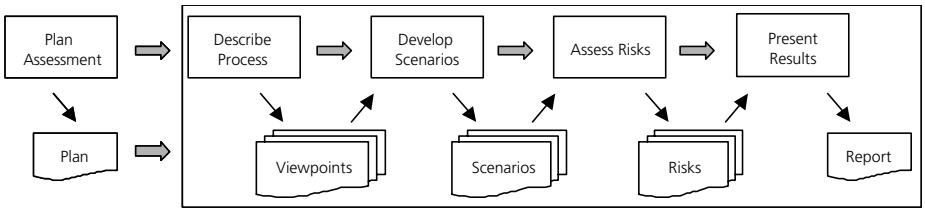


Figure 6. Assessment method overview

3.1 Assessment Planning

The purpose of this phase is to set the constraints of the assessment and find relevant interviewees. The only input to this activity is the client of the assessment. The assessor gives the client an outline of the assessment plan for resource estimation, and a description of what the assessment will result in. Thereafter the assessor and the client agree upon a purpose, a scope, a timeframe, and a system of metrics for the assessment.

The purpose is used to focus the assessment, and the scope is used to further narrow the assessment to e.g. a particular type of subproject. The timeframe is used to narrow the applicability of the assessment in time, to be able to formulate more concrete questions. The system of metrics, the scales for cost and probability, is used to collect data from the interviewees about the cost of supporting the scenarios that will be developed, and to determine the probability of these scenarios. The cost of supporting a scenario is regarded as the cost of maintaining the same degree of support although the process environment changes due to that scenario.

Finally, interviewees for the assessment are selected in order to be able to plan the assessment. The context description developed in the following subsection is used to identify the roles in the organization that would give the most

valuable information. Therefore, these two activities are carried out in parallel. The interviewees shall cover different aspects of the process and organization, limited by the scope.

3.2 Context Description

The purpose of the context description is to identify roles in the organization that are suitable for interviews, and to create a starting point for open-ended interview questions.

In order to do this, a process description, an organizational structure description, and a list of roles and their responsibilities shall be created. The vocabulary of the organization can also be useful, in order to simplify the interviews.

3.3 Scenario Development

The purpose of this phase is to develop a set of scenarios that will constitute the basis for the risk assessment. The input to this phase is a set of interviewees and an initial knowledge about the process and organization. The assessor develops scenarios from issues discussed with the interviewees.

The interviews are based on the initial knowledge of the process and the purpose of the interviews is to elicit and discuss issues that affect the current situation in the organization, see Figure 7. The activities of the process and the different parts of the organization can be used as an initial set of such issues. The set of issues is expanded during the interviews.

Ideally the interviewees provide the assessor with scenarios, but most likely the assessor will have to create scenarios from the issues discussed. Analyzing how the issues were brought up provides clues as to how they are related. From the issues a set of *abstract scenarios* is sketched. Each issue shall be covered, but several issues may affect an abstract scenario due to their relations. When the issues are covered the abstract scenarios are specified, resulting in *concrete scenarios*, see Figure 7. An abstract scenario may then result in several concrete scenar-

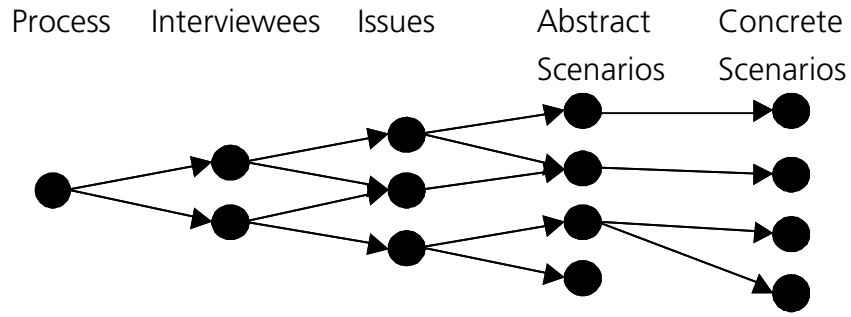


Figure 7. Scenario development: The context description provides suitable interviewees. The outcome of the interviews is a set of issues. Abstract scenarios are elicited from the issues. These are, if possible, specified to more concrete scenarios

ios. Concrete scenarios shall not contain too long chains of events, as it becomes more unlikely that the organization would take actions according to that chain of event. In such a case it may be possible to split the scenario. Some abstract scenarios may be hard to specify, due to lack of domain knowledge. If these abstract scenarios are included they introduce a higher risk of interpretational errors.

The set of scenarios shall then be reviewed in order to see if it is possible to answer whether the scenario is likely and costly. The *engineering environment factors* presented in the following subsection should also be relevant to the set of scenarios.

3.4 Risk Assessment

The purpose of this phase is to assess the risks of the scenarios and draw conclusions. The input to this phase is the set of scenarios and the set of issues from which the scenarios were developed.

A second round of individual interviews lets the interviewees state the probability of each scenario and the cost of supporting each scenario according to the scales determined in the assessment-planning phase. The participants shall be given the option not to answer a question if they feel they have no reasonable answer. The method is however designed to only give an indication of the process-related risks that the organization faces so no exact answers are required.

The interviewees shall also state which *engineering environment factors* the scenarios affect, in order to simplify scenario interpretation. A suggested set of factors is: Process, Resources, Organization, Competence, Software Architecture, Technology, Tools, Process Implementation and Management. This set of factors is not firmly defined and many have to be reinterpreted and adapted to the specific situation. The purpose of it is to be able to draw conclusions that are more general than those who only concern a specific scenario. The generalizability of conclusions drawn from scenario-based methods is further discussed in Section 6.1.

After the second round of interviews each scenario can be given a risk value. This can be done in a number of ways. The basic principle is to multiply the cost of a scenario with the probability of a scenario. This is exemplified in the case study.

If the interviewees disagree to a large extent, the calculated risk values may be misguiding. More than one technique of calculating them can then be used to determine an appropriate set of high-risk scenarios. The level of agreement per scenario can be analyzed with the dispersion of the risk values, which can be calculated as the standard deviation divided by the mean risk for that scenario (Regnell *et al.*, 2001).

The subjects' level of agreement can be investigated with a Kappa analysis (Fusario *et al.*, 1997). It compares the answers given by each possible pair of interviewees. The analysis method requires that the answers are divided in a few discrete steps, which puts constraints on the selected system of metrics. This

means that the method produces two values of agreement for each pair of interviewees, one for the costs and one for the probability of all scenarios.

With a Kappa analysis a matrix is built, where each cell (x,y) contains the number of times interviewee one gave an answer x when interviewee two gave an answer y. Elements on the diagonal contribute to a higher Kappa index, indicating that the participants agreed completely, and results that are close to the diagonal can contribute according to a weight given by Fleiss and Cohen (Fusario *et al.*, 1997). A Kappa of above 0.21 is considered 'fair', above 0.41 'moderate', and above 0.61 'substantial', according to a rule of thumb suggested by Landis and Koch (Fusario *et al.*, 1997).

The proposed assessment method has now produced an appropriate number of high-risk scenarios that the organization could focus on in future process improvement initiatives. This set of scenarios is however very specific, but can be generalized by analyzing the interview issues from which the scenarios originated.

Another generalization that can be made is to study the engineering environment factors that the high-risk scenarios affect. For each scenario the participants should have stated which engineering environment factors are most affected. These can be analyzed by counting how many times each factor was mentioned by the interviewees and weighting this against the risk value of the scenario in question. Dividing this by the total risk value of all scenarios gives a set of indices between zero and one. These indices suggest how much each factor affects the total risk on the organization, and shows which factors should be prioritized in risk-reduction efforts.

The output of this phase is a set of high-risk scenarios, related to a set of organizational issues and a set of prioritized engineering environment factors. If the level of agreement between the interviewees has been analyzed, information about e.g. communication flow can emerge as a side effect.

3.5 Assessment Presentation

The purpose of this phase is to present the outcomes of the previous phase to the clients and the participants of the assessment in an understandable way. It is important to let everybody that has dedicated time and effort on the assessment give and get feedback. This feedback is part of the presentation phase, and it can be valuable for strengthening the conclusions from the assessment.

4. Case Study

An initial evaluation of the method was performed on a software process used in a software engineering course including an industry-like project (Wohlin & Regnell, 1999). The purpose was to evolve the procedures of the method. The results were guidelines for the interviews of the proposed method. In order to evaluate the resulting method a case study was planned and performed at Company A, presented in Chapter III.

4.1 Evaluation Planning and Operation

The process was studied resulting in a context description, and the assessment was planned with the Process Developer at Company A. The assessor and the Process Developer decided on a timeframe including the current and the following platform project. Table 9 shows the system of metrics for data collection that was selected.

Table 9. Estimate scales

Value	Scenario Probability	Scenario Cost
1	Never	Already supported
2	Unlikely	Inexpensive
3	Likely	Expensive
4	In all likeliness	Impossible

Four roles were selected as interviewees, to cover as much as possible of the requirements process. The Process Developer gave a presentation of the process and organization, which was the base for the context description. From this description the Process Developer, the Project Manager, a representative of the System Designers, and a Subproject Manager responsible for a module with new functionality, were selected as interviewees.

In the first set of interviews a set of issues was developed. The first interview was held with the Process Developer, and it was based on a set of generic questions regarding the roles and responsibilities of the interviewee, and a small set of issues found when describing the process. The interviews with the other three participants followed the same basic procedure, but were supported by a larger set of issues. The interviews were recorded on tape in order to easier find abstract scenarios during the following analysis of the issues. The interviews lasted less than one and a half hour, and apart from the assessor and the interviewee, the Process Developer was also present at all interviews.

The interviews resulted in 26 issues, listed in Table 10. These were ordered in an Entity-Relation diagram and all that had been said regarding each issue was assembled under each issue.

This led to a set of abstract scenarios for each issue. Some of these covered several issues, and they were all traceable back to their original issues. It was realized that many of these scenarios would be hard for the interviewees to assess. One of the strengths of scenarios is to make an abstract phenomenon more concrete. A risk is however that the scenarios become too narrow, and that closely related scenarios do not result in similar values for probability and cost. Therefore both abstract and concrete scenarios were provided during the second interview, where the concrete scenario acted as an example among others within

Table 10. Interview issues

Reuse	Architecture	Platform Vision	Communication
Platform vs. Product	Architectural Changes	Requirements Quality	Technical Evolution
Integration	Testing	Market	Subcontractors
Resources	Line vs. Project	Tools	Organization
Process	Estimates	Metrics	Competence
Decision Support	Requirement Changes	Requirement Sources	Organizational Culture
Business Focus	Responsibilities		

that abstract scenario. Some abstract scenarios were however found to be too broad and had to be split up into several concrete scenarios. Every scenario was not made concrete due to lack of domain knowledge. The result was a set of 36 abstract scenarios, whereof 31 were exemplified with concrete scenarios. Table 11 shows some of these scenarios. Issues with scenario development, such as these, are elaborated in Section 6.1.

During the second set of interviews the participants were asked to assign values for probability and cost of each scenario according to the scales above, and determine which engineering environment factors where affected by the scenario. The original list of factors was *process*, *organization*, *competence*, *resources*, *architecture* and *technology*. It was derived from the issues and determined by the assessor and the Process Developer. During the first and second interview the list was appended with three additional factors that emerged during the interviews: *tools*, *process implementation* and *management*.

Table 11. Sample scenarios

Abstract Scenario	**Concrete Scenario**
Applications must be rebuilt after an architectural change.	The architecture must be rebuilt due to new performance requirements, which means that the interfaces to most applications must be rebuilt.
Problems during integration of the second product based on the current platform.	Performance bottlenecks do not emerge until building the second product on this platform.
Products set quality requirements on the platforms.	The first product to be built on this platform demands that X% of all branches in code shall have been run in test.
Project managers get a larger resource responsibility.	The line organization functions as a consultancy firm for the project organizations.

During the first interview, which was held with the Process Developer, interpretation problems were found with three scenarios. These were clarified and caused no problem. During the second interview a scenario was however found that had to be split in two. The scenario concerned architecture and the System Designer realized that two similar problems called for different solutions. This was confirmed by the two remaining interviews.

The Process Developer did not attend the other interviews during the second round of interviews. The interviews were recorded on questionnaires, and they took less than an hour each.

4.2 Evaluation Data Analysis

Since the assessment method is qualitative in nature, it is not possible to draw significant quantitative conclusions from the results, as discussed in Chapter II. The assessment is intended as a tool for decision support for future process improvements, indicating problem areas in the process and organization.

Since the interviewees gave intermediate values during the second round of interviews, the scales found in Section 4.1 were modified to include seven steps; the four original and three intermediate steps. These steps were numbered from 0 to 6, as it was natural for both the probability and the cost scale to start at zero.

4.2.1 Risk Assessment

The risk values for each scenario were calculated by assuming that the original values for cost, c_{is}, and probability, p_{is}, that an interviewee i gave for a scenario s were dependent. The values were assumed to be dependent, because when the interviewees studied a scenario they in most cases automatically assessed the risk in some way. This was noted by frequent phrases such as "we will not be able to support this scenario but it doesn't matter, because it will never happen" or "we are already working this way so the scenario has already happened and it will cost nothing to support". Furthermore the scale for cost was assumed to be interpreted as an exponential scale.

The second assumption lead to a transformation of the cost scale from the initial range of 0 to 6, to scores ranging from 1 to 10^6, giving the cost c'_{is} for one scenario and one interviewee:

$$c'_{is} = 10^{c_{is}} \quad (1)$$

The probability scale was normalized, ranging from 0 to 1, giving the probability p'_{is} for one scenario and one interviewee:

$$p'_{is} = p_{is}/6 \quad (2)$$

After these transformations the first assumption was considered, and the risk value for each scenario was calculated accordingly. The risk, r_{is}, that each inter-

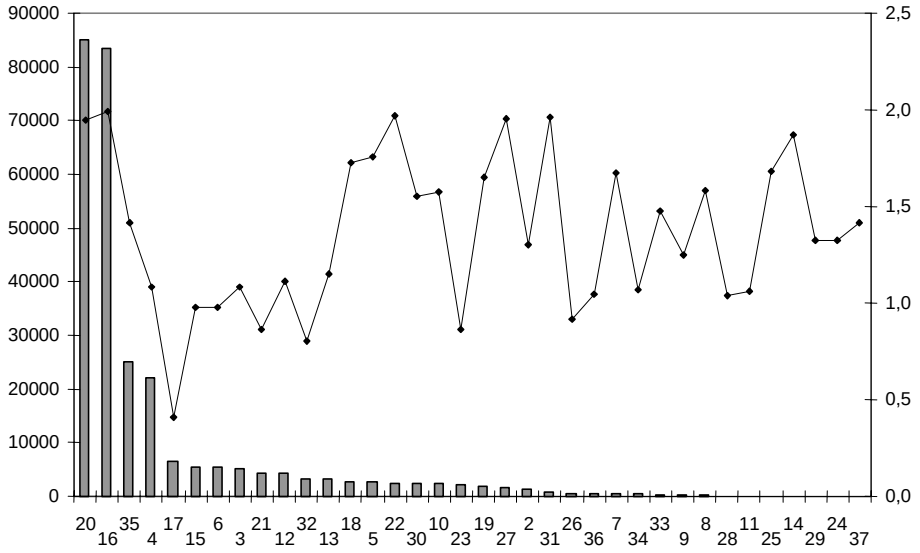


Figure 8. The scenarios sorted by their risk calculated according to the proposed method

viewee gave each scenario was calculated by multiplying the resulting cost with the resulting probability:

$$r_{is} = c'_{is} \cdot p'_{is} \quad (\mathbf{3})$$

This resulted in two to four values of risk for each scenario, depending on how many interviewees had assigned a cost and a probability value to each scenario. Finally the average risk r_s for each scenario was calculated from the individual risk values:

$$r_s = \sum_i \frac{r_{is}}{n_s} \quad (\mathbf{4})$$

In Formula 4, n_s is the number of answers for scenario s.

Figure 8 shows the scenarios sorted according to their risk calculated according to the *exponential dependent technique* described above, with the risk value on the Y-axis and the scenario number on the X-axis. The line of diamonds shows the dispersion of the risks of the scenarios, here measured as the standard deviation divided by the average risk. The four first scenarios show much higher risk than the following, which indicates that they shall be prioritized. When analyzing them more carefully it however became clear that the participants gave very varying answers, which is also indicated by the high dispersion of the two scenarios with highest risk.

4.2.2 Participant Interagreement

The high dispersion of risk values for some scenarios called for further analysis of the level of agreement between the participants. A Kappa analysis (Fusario *et al.*, 1997) was made on the interviewees' values of cost and probability. Kappa for the costs ranged from -0.02 to 0.34 among the possible pairs of interviewees, with an average pair agreement of 0.20. Kappa for the probabilities ranged from 0.13 to 0.42 among the possible pairs of interviewees, with an average pair agreement of 0.25. The low levels of agreement can be explained by that the interviewees were selected to get as differing views on the process as possible.

4.2.3 Other Risk Calculation Techniques

Since the risk values for the scenarios were uncertain, other ways of finding the most important scenarios were investigated. Firstly, the scale for the costs transformation was studied. If the original, linear values were used, the order of the scenarios differed. It was however considered reasonable that the original verbal scale was not interpreted as linear by the interviewees, but rather exponential or worse. The *linear technique* of assigning risk values was later used as a reference when evaluating the assessment method during the feedback session.

An analysis was made as to whether the order of the scenarios was sensitive to the base used in the original exponential transformation. The original base was ten, i.e. each of the six steps in the scale meant a tenfold increase in cost. A twofold increase was tried, and the order of the scenarios differed slightly. The four most important scenarios were however still the same. If a fourfold increase or larger was used, the ranking order was identical to the tenfold increase.

The scenarios were also sorted by selecting the answer giving the highest risk for each scenario, i.e. saying that if anyone can sense a risk in a scenario, there is a risk. This order, the *maximum-risk technique*, was somewhat different as it gave a large number of ties. The four first scenarios were however still the same as the four first of the *exponential dependent technique*. The scenarios that each individual rated highest were also examined. None of the interviewees ranked the same scenario highest and these four scenarios were identical to the four most important scenarios in the *exponential dependent technique*.

The last examined method to calculate the risks of the scenarios was to assume that the values for cost and probability of the scenarios were independent. The exponential transform was still applied to the costs for each individual answer. Then the average cost and average probability for each scenario was calculated, over the number of answers for that scenario. These two averages where then multiplied to give a value for the risk of that scenario. This technique, the *exponential independent technique*, implies that the participants, when studying one scenario, gave the answer for cost independently of the answer for probability. This would perhaps be beneficial, but was not the way in which the answers were given. Using the *exponential independent technique*, two of the four most important scenarios were found among the four most important scenarios from the *exponential dependent technique*.

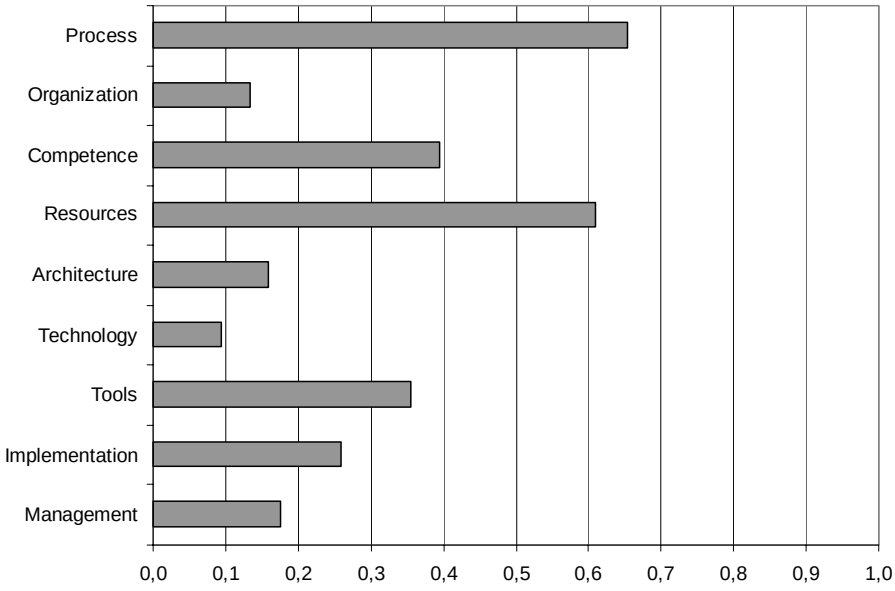


Figure 9. A diagram showing the impact each of the engineering environment factors has on the risk of all scenarios

The *exponential dependent technique* was selected as the prime candidate for calculating risk in this particular case, a choice that was confirmed at the feedback session. Most of the following analysis is based on this assumption.

4.2.4 Analysis of Engineering Environment Factors

The nine engineering environment factors could now be analyzed based on the calculated risk values for the scenarios. The sum of the risk values for all scenarios where one of these factors had been brought up by an interviewee was calculated. This value represents the risk that affects that factor. By dividing it by the total risk for all scenarios the percentage of the organizations risk affecting that factor was calculated. This analysis is visualized in Figure 9. 'Implementation' in the diagram means process implementation, while the other factors are the same as in Section 4.1. The sum of the affected factors is more than one, since several factors appear in a single scenario.

The figure shows that 65 % of the discovered risk somehow affects the process, and that 61 % affects the resources. Therefore it would be recommended to focus on these two factors when trying to reduce risks according to the results of this assessment.

4.3 Evaluation Results

The analysis gave clues to which scenarios were most important. Because of the low level of agreement between the interviewees, three techniques of calculating

risk were considered. The techniques were: *exponential dependent*, *exponential independent* and *linear*, as discussed in Section 4.2. The four highest-risk scenarios from each technique were selected, which resulted in nine unique scenarios. These were compared to the interview issues, and three groups of scenarios emerged. These were:

- *Architectural issues* related to rebuilding the architecture due to e.g. new technologies, and architectural problems with system integration.
- *Resources and competence*, which are lacking if Company A has to react to the high-risk scenarios or introduce new technology.
- *Process issues* mostly regarding the relation between requirements and test.

Among these issues, process, resources, and to some degree competence and tools, had already been pointed out by the previous analysis of engineering environment factors. One of the nine scenarios did not fit in these groups, as it only affected the issue of subcontractors. This scenario was also regarded as least important of the nine by the three selected risk calculation techniques. These results were presented to the interviewees on a feedback meeting.

4.3.1 Feedback Session

A meeting was held with the participants in order to present and verify the results. An exercise was conducted in order to select the most appropriate technique for risk calculation. The participants were given three sets of high-risk scenarios, consisting of the four scenarios with highest risk according to the three techniques *exponential dependent*, *exponential independent* and *linear*. They were then given the question: If you were to start a process improvement initiative, which set of scenarios would you base it on? All four participants selected the four scenarios given by the *exponential dependent technique*. This was a verification of the assumptions made during the risk assessment, and the presentation that followed was based on the selected technique.

The results above were presented and the participants agreed to the risks associated with resources and the issue of improving the connection between their requirements phase and test phase. They showed interest in analyzing their low level of agreement, and agreed on letting the Process Developer analyze the data from that aspect. They were furthermore satisfied with the level of resources they had spent on the assessment. Together they had spent less than 30 working-hours, where the Process Developer had done the main part. A presentation of the whole case study for middle and senior management was scheduled.

4.3.2 Threats to Validity

This subsection covers the threats to validity that emerged during the assessment operation.

The Process Developer had participated during the first set of interviews. This hindered anonymity, but the discussions were open and the other interviewees were not afraid to criticize the process or other aspects of the organiza-

tion. This decreases the threat that important issues were not brought up during the interviews.

A major concern is the quality of the scenarios. A high-quality set of interview issues is a base for scenario development, as is the assessor's knowledge of the organization, found in the context description. The assessor's ability to develop an adequate set of scenarios from this material is however a decisive factor. The method gives support but training and practice is also essential. One way of validating the scenarios is to analyze future changes to process, architecture or structure of the organization, and the causes for these changes.

A threat during the second set of interviews was that all interviewees except one used intermediate values, perhaps because words like 'never' and 'impossible' are awkward to assign to events. The interviewee who did not use intermediate levels also filled in the questionnaire himself, which the others did not, and he assigned factors to scenarios after he had assigned all cost and probability values. The impact of such threats is hard to assess.

From a correct set of high-risk scenarios the identification of high-risk issues should present no threat. The generalization from high-risk scenarios to prioritizing engineering environment factors however contains a number of threats. Firstly the factors were not thoroughly defined and were left for the interviewees to interpret. Secondly the set of factors was created informally and important factors may be overlooked. These threats could be avoided with a better defined set of engineering environment factors. In this study, the factors have however been emerging, as a part of the conclusions drawn in Section 6.2.

5. Results and Method Improvements

This study has introduced the concept of process flexibility. A method is proposed which assesses process flexibility by analyzing risks within an organization and its process. The method uses scenarios in order to focus on the particular situation at an organization. The method is evaluated in a case study, where the process flexibility at Company A has been assessed. The method has proven promising in the aspects of cost, feasibility and effectiveness:

- **Cost**. During the case study, the assessed organization provided three participants who spent less than 30 working-hours in total. Of these 30 hours the contact person at the organization spent about a third. The assessor spent around 60 hours collecting and analyzing data. The cost of performing an assessment of this type is therefore considered low.
- **Feasibility**. The low cost also leads to increased feasibility, as there were no major problems in scheduling appointments with the participants. An assessor using this method will need some training, most of which can be derived from this article. With a trained assessor the method generates questions and scenarios that are possible to answer and assess unambiguously. The results generated are also easy to interpret and feasible to act upon.

- **Effectiveness**. The method has in the case study produced results that the participants agreed described the situation at the organization. Three areas were identified that called for improvements.

The method has only been used once, and has many opportunities of improvement. From a researcher's point of view, the method also has opportunities of generating general conclusions about process flexibility. To further increase the feasibility of the method the guidelines and procedures of the method could be improved, either from experience with the method or from lessons learned from other software process assessment methods.

The effectiveness of the method could be validated by following up the situation at assessed organizations. One possible way of doing this is to see if any of the scenarios has occurred in the timeframe specified in the assessment. In order to make generalizations from this method it must first of all be possible to compare results from different assessments. If this is possible one can eventually compare organizations that are flexible, according to the method, versus organizations that are inflexible. This could provide general conclusions as to how organizations and processes should be designed in order to be flexible.

Further studies of the proposed method may provide additional insights into ways of understanding and improving scenario-based process flexibility assessment, which hopefully can contribute to an effective and efficient method for risk management in software development.

6. Discussion

This study has raised two topics of discussion. One is the complexity of scenario generation and interpreting results from scenario-based methods. The other is the relation between a software-developing organization, its products, processes, and the architecture on which their products are based. These issues are discussed in the following.

6.1 Scenario Generation and Interpretation Revisited

The method presented in this study is based on scenarios, as is SAAM (Kazman, 1994) and ATAM (Kazman, 1998). The real options methods presented in Chapter V can also be said to be based on scenarios, as the set of likely requirements changes actually is a set of future scenarios. Other forms of evaluation and assessment are analytic methods, such as the one presented in Chapter X, and methods based on checklists and standard forms such as CMM (Paulk *et al.*, 1993).

The upside of scenario-based techniques is that they make abstract and unexplored phenomena quite concrete. The scenarios will also be relevant to the studied context, as they are developed in cooperation with stakeholders, rather than a pre-fabricated standard evaluation form. To exploit these benefits, it is however required that the set of scenarios is adequate, i.e. representative of future outcomes of the phenomena under study. This study however shows that it

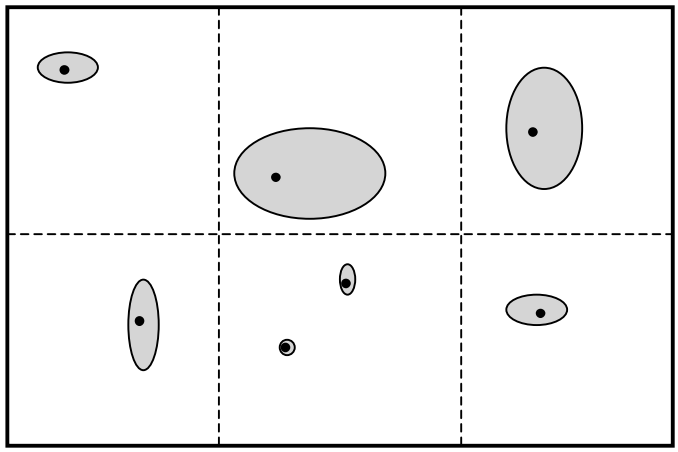


Figure 10. Space of outcomes for scenarios. The areas indicate the difference in outcome that can result from an abstract or underspecified scenario, points indicate the specific scenarios that are sought for, and the dashed lines demarcate a classification of six types of scenarios.

can be difficult to make stakeholders generate scenarios, an experience shared from case studies with the CBAM method (Moore *et al.*, 2003). Most scenario methods therefore require assistance from method experts such as external consultants performing the evaluations.

This study has also uncovered a number of theoretical issues with scenarios. A scenario represents an area of likely and similar outcomes in the space of possible outcomes. This area is larger or smaller depending on the level of abstraction of a scenario, as shown in Figure 10. But if the scenario is not close to a point in this outcome space, how can we assess the probability and effect of the scenario? These will not be the same throughout the space of an abstract scenario, although hopefully similar. These differences are shown in Figure 11, an example of how the outcome in e.g. risk of the top three scenarios of Figure 10 could behave. So close-to-point scenarios are easier to assess the risk of. But the whole set of scenarios is supposed to tell us something about the future – some type of average future. How can we know that the set of scenarios is representative of the future, so that our set of scenarios will tell something about also the effects of most other scenarios? The set of scenarios can be classified, so that we can generate scenarios from all possible classes, and get our coverage more evenly distributed. This is shown with the dashed lines in Figure 10. But we then have to assume that the behavior of the scenarios is somewhat linear, and how can we assume this? Why can't relatively small changes in a scenario result in quite different outcomes? And how does the space of outcomes "between" scenarios behave?

Examples of how the outcome of a scenario could behave nicely, or not so nice, are shown in Figure 11. The middle example shows a category where we would have to generate many scenarios, and make them very specific to be able to describe the behavior of scenarios of that category. In the right-most case, we

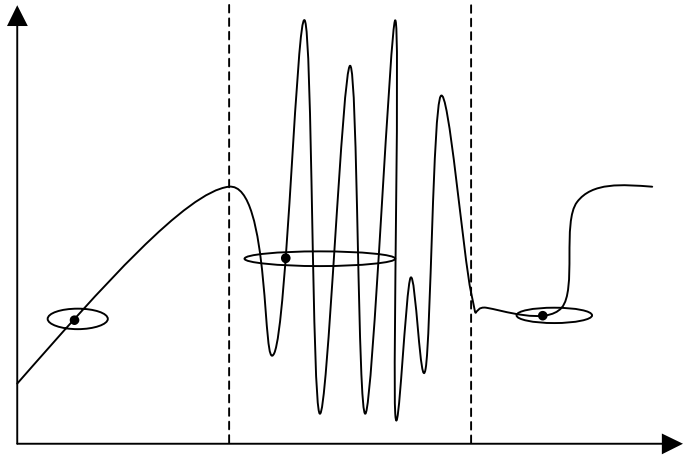


Figure 11.Possible outcomes of scenarios. Points indicates the sought for, fully specified scenario, areas show the spread of the top three underspecified scenarios of Figure 10. The x *axis shows one dimension of Figure 10, while the* y *axis shows outcome, which in the case study would represent risk.*

could describe the behavior quite well with only a few extra scenarios. In some cases scenarios can be parameterized, as to some extent is done in CBAM (Moore *et al.*, 2003). The leftmost example in Figure 11 could e.g. be parameterized. Most often there are not only quantitative differences between two scenarios, but also qualitative. This is when the outcome-space between scenarios is difficult to predict.

Abstract scenarios, which take up a large area in the outcome space, will also be difficult to compare to point-shaped concrete scenarios. For the probabilities to be comparable, the scenarios should be on the same level of abstraction. Kazman *et al.* (1994) introduce two levels of abstraction, or categories, when they separate usage scenarios – those that show examples of how end users will try to use a system – from change scenarios – those that describe change requests from users and customers. America *et al.* (2005) further introduce strategic scenarios, from marketing literature, which describe how the market and economy in general might evolve. These strategic scenarios might lead to change scenarios, and when evaluating an organization's flexibility with regard to a set of scenarios, these scenarios should be of the same abstraction level.

Another problem is that scenarios are short chains of events, or stories. Sometimes a scenario only consists of one event, but more often they involve assumptions and several dependent and independent events. A longer scenario will then be more unlikely than a scenario based on one single event, which makes the comparison of probabilities even more difficult. The evaluator could in these cases raise the responses to the same level of abstraction, but some of the directness of the scenario-based approach is then lost. These issues call for further work on scenario-based methodology.

6.2 Development in a Line-Oriented Fashion

Development at Company A is performed in projects, most of which result in a new product being released for production. In the studied case the project developed a platform to be used by product-oriented projects. Practically all development resources are however consumed by these projects, and less prestigious line-oriented work such as quality or maintenance has difficulties penetrating through to the projects.

A quite different approach was taken by Company B, presented in Chapter III. A case study has been performed on the development process, and specifically the requirements process at Company B. The REPEAT process is a result of an improvement program that started in 1995, as Company B considered efficient requirements engineering a key success factor. After the introduction of REPEAT, a significant improvement in delivery precision and product quality was gained. However, after a number of releases with REPEAT, it was realized that market pressure resulted in a number of further challenges regarding throughput and congestion (Regnell *et al.*, 1998). Therefore a process simulation model was developed to analyze bottlenecks in the requirements process (Höst *et al.*, 2001).

6.2.1 The REPEAT Process

REPEAT manages requirements continuously by controlling a product pipeline in which three releases are developed in parallel. The product pipeline delivers two new product releases per year. REPEAT covers typical RE activities, such as elicitation, documentation, and validation, and the process has a strong focus on requirements selection and release planning. A schematic picture of the process is shown in Figure 12.

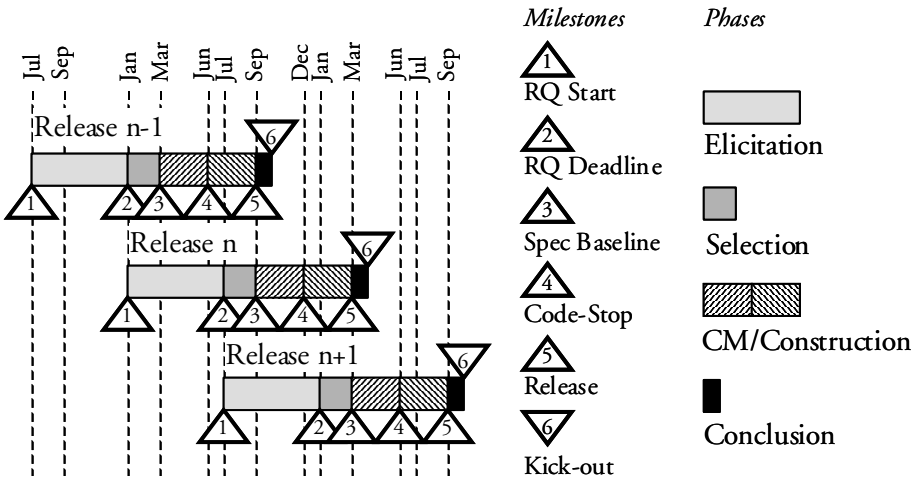


Figure 12. The milestones and phases of the REPEAT process, aligned with a fixed release schedule

REPEAT is instantiated for each release, and each process instance has a fixed duration of 14 months. Each REPEAT instance consists of five different phases separated by milestones at pre-defined dates. The Elicitation phase deals with the collection and initial classification of requirements. The Selection phase includes detailed specification of each requirement and release planning. The Change Management phase is active in parallel with construction (design, implementation, and testing of requirements for the coming release) and manages changes in requirements priorities due to events such as emergence of high-priority requirements and delays. The Conclusion phase includes post-mortem documentation. Each phase is further described below.

Elicitation: The elicitation phase includes two activities: collection and classification. Collection of requirements is made by an issuer that fills out a web-form and submits the requirement for storage in a database which was developed in-house. Requirements are described using natural language and given a summary name by the issuer. An explanation of why the requirement is needed is also given. The issuer gives the requirement an initial priority P, which suggests in which release it may be implemented. P is a subjective measure reflecting the view of the issuer, and is measured on an ordinal scale with three levels, as shown in Table 12.

Table 12. The ordinal scale of the priority P

Priority	Semantics
1	The requirement is allowed to impact ongoing construction of the previous release.
2	The requirement is incorporated in the current release planning.
3	The requirement is postponed to a later release.

Selection: The goals of this phase are: (1) to select which requirements to implement in the current release, (2) to specify the selected requirements in more detail, and (3) to validate the requirements. The output of this phase is a requirements document that includes a selected-list with a detailed specification and effort estimation in hours of all selected requirements, and a not-selected-list including the requirements that are postponed to the next release. The selected requirements are divided into a must-list and a wish-list. The must-list comprises requirements that are estimated to take 70% of the available effort, while the wish-list comprises requirements that are estimated to take 60% of the available effort. This implies that if the effort estimations are correct, half of the wish-list will be implemented, and the rest will be reconsidered for implementation in the next release. However, all the requirements on the wish-list are specified, so if the estimations are not correct there will still be a number of specified requirements to implement in the release.

Change Management during Construction: This phase of the REPEAT process is carried out in parallel with the design, implementation, and testing of the requirements, and handles changes in the priorities of the requirements. There are two sub-phases of this phase, one before the code-stop milestone (3-4 in Figure 12) and one after code-stop (4-5 in Figure 12). After code-stop no implementation is carried out. Instead the focus is on testing. If new priority-1-requirements are issued, these may be allowed to affect ongoing construction, and in the change management phase the requirements on the must- and wish-list may be rearranged so that new and more important requirements can be incorporated. The 70%– 60% rule for the must- and wish-lists must, however, still hold, implying that some less important requirements should be postponed in order to incorporate the new, more important, requirements.
Conclusion: In this phase metrics are collected and a final report is written that summarizes the lessons learnt from this REPEAT enactment.

6.2.2 The Process Simulated

During 1998 and 1999, the number of unimplemented requirements in the requirements database was increasing, and the REPEAT process had at times been in a state of congestion. Process simulation was carried out to analyze how to improve resource allocation to reduce this congestion. The simulator was event-based (Banks, 1996), and modeled on a queuing network (King, 1990). The structure can be seen in Figure 13, and shows how development was modeled as a series of consecutive but parallel releases divided in 3 development phases focused on requirements engineering. Process simulation presented quantitative process improvement proposals, but foremost aided in understanding how Company B managed to develop and release products regularly, without the use of proper development projects and the associated overhead.

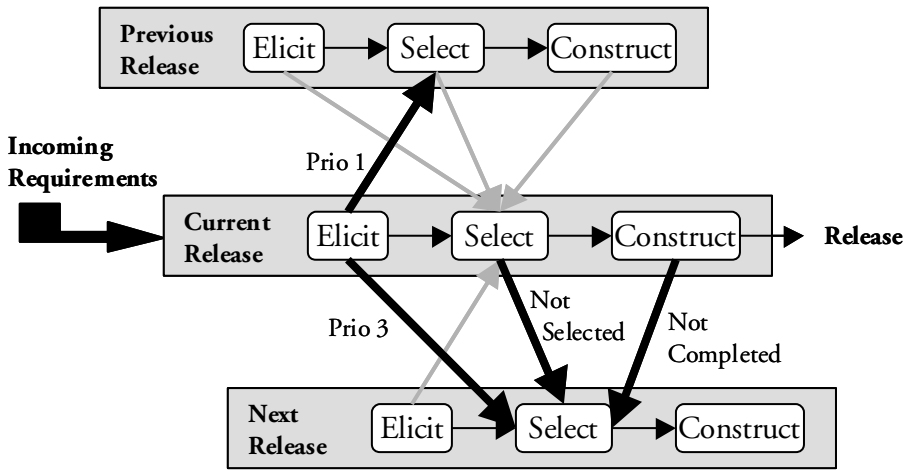


Figure 13. Simulation model

6.3 A Framework of Process, Organization and Architecture

With the introduction of engineering environment factors, this study has shown relations between the software process, organization and architecture. An organizational framework has emerged from this study along with the analysis of the context of the companies studied in this thesis – especially by studying the differences between project-oriented organizations such as Company A, and line-oriented organizations such as Company B. Two instantiations of this framework are shown in Figure 14. The main components of the general framework are the organization, how work is done, and the products that are produced. These components were identified from the environmental factors presented in the flexibility assessment case study performed at Company A.

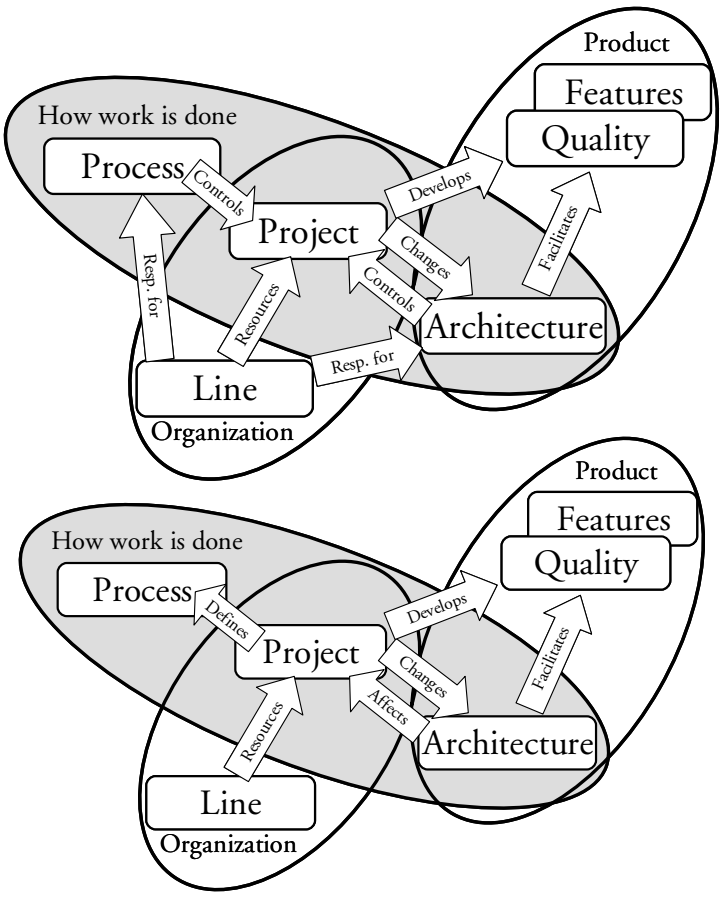


Figure 14. Organizational Framework. Above - organization with strong line management. Below - organization with strong projects

The organization consists of the hierarchy of line management and project management, often combined in a matrix organization. How work is done is controlled by the process in use, the project organization and project model, and by the architecture. The product consists of features and qualities, and of the architecture, which enables feature content, and most importantly, enables the desired quality attributes. The arrows in the framework indicate which sub-components control each other, and the direction and strength of these relationships may vary in different organizations, especially with regard to the division of responsibility between line and project management.

The framework instantiation shown in the upper part of Figure 14 shows an organization with strong line management, responsible for both a formal process and an established architecture, which both controls how projects are performed, and how a project can change the architecture. This is comparable to the Company B situation described in Section 6.2, where the products were developed in an assembly-line fashion, rather than through proper projects. The lower part of the figure shows an organization where most resources are consumed by typically one single project, which defines both architecture and process. From the case-study presented here, Company A would resemble the lower part of Figure 14, as the organization is quite project-driven, but the concept of a platform and platform projects means that there is a line function with decision power and independent resources.

The general framework highlights that even though the architecture is a part of the product, it also affects how development is carried out. This has implications for the special problems of architectural change, as shown in Chapter VII. The framework can also be used to visualize the balance of power in a company: either as a company where line management has responsibility for long term process and architecture goals, and power to enforce them; or as a company where resources and control is focused on projects and short term product development.

VII The Architectural Change Process

Software architecture is recognized as a critical factor for successful products, but few have studied how organizations decide on architectural changes. In this chapter the topic is studied through several cases. The changes are in all cases changes to the quality attributes of the system, and follow the same general process. The study shows that architectural changes have aspects of both functional and organizational changes. An architectural change does not only need to be technically sound, but it also needs to be anchored firmly in the organization. This chapter describes the general architectural change process, and gives both architects and managers guidelines to balance short-term project goals and long-term organizational goals within this process.

From Nedstam, J., Karlsson, E.-A. and Höst, M., "The Architectural Change Process", Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04)*, pp 27-36, Redondo Beach, CA, August 2004.*

1. Introduction

Software architecture is becoming a well-established field in technical terms, i.e. different types of architectures have been characterized (Garlan & Shaw, 1996); different useful views of the architecture have been described (Soni *et al.*, 1995; Kruchten, 1995); and books have been written, covering the general area, e.g. Bass *et al.* (1998), Hofmeister *et al.* (1999) and Bosch (2000). However, little research has been done on how decisions on architectural changes are made in organizations.

Architectural changes are often different in nature from functional changes. They can impact larger parts of the product, and they are expensive to implement. In this study it is also observed that architectural changes have a varying degree of organizational impact. Functional changes often originate from a customer demand and are the responsibility of a defined role in a company, i.e. product management. Architectural changes can on the other hand emerge from various sources, and roles are seldom defined to drive such changes.

This work recognizes that any change to a product can be classified anywhere on a continuous scale from solely architectural changes, affecting structure rather than user-observable attributes, to solely functional changes, limited to a single user function. Comparisons made between architectural and functional changes will mostly consider these two extremes, and the changes studied are considered to be close enough to the architectural end of this spectrum to support such arguments.

The process for making decisions regarding functional changes and features has received attention in recent years (Evans *et al.*, 1997). Software development processes generally support this well, for instance through the stage-gate approach (Cooper, 1990; Cooper & Kleinschmidt, 1993). When it comes to decisions regarding the software architecture, the architect is often not so well supported, neither for the analysis of the technical solutions, nor for the organizational impact of the change.

The practitioners who contributed to this study needed help in managing architectural changes, help that was not provided from surveying literature. An initial research question was therefore posed: *How are architectural decisions made?* In order to answer this question a qualitative study has been performed on how changes to the software architecture have been handled at three software development organizations, and what forces drive the need for changes and control which solutions are decided upon. The research question was therefore refined to: *What is the implicit or explicit process of architectural change?*

Based on the case studies, such a general process has been identified. Detailed investigation of the cases, by comparing the identified process to the process for deciding on functional changes, and theories of organizational change, led to suggested improvements and guidelines for each step of the process. The validity of the process and guidelines has then been analyzed through a workshop session and through a reference case study performed at Company E. The reference case is presented in Section 6 and visible at the bottom of Table 13.

This introduction is followed by a description of how this study was conducted. Section 3 provides the case descriptions, and the suggested process is presented in Section 4, followed by a comparison between cases and process. Section 6 describes the validation, followed by conclusions.

2. Conduct of Research

This is a study of seven architectural changes initiated at three software-developing companies. All companies develop products for a mass-market, and their products have long lifetimes. Therefore their architectures need to support several simultaneous versions of their products, with several releases over an extended period of time. The research was exploratory in nature, and the overall research question related to architectural decision making.

Qualitative data has been collected in two sets of interviews. The first set was held with architects and system designers at the three companies to collect in-

formation about the companies, their products, and their architecture. The interviews were open, and through the three interviews suitable architectural changes were identified. Changes were selected that affected varying aspects of the architectures to a small or large extent. The study was also constricted to cases that were recent in time, so key personnel and documentation could still be found.

Key persons in those changes were interviewed in a second set of interviews. This set of seven sessions, involving from two to five respondents, was guided by these interview questions:

1. What is the architectural change?
2. Why was the architectural change needed?
3. Who initiated it?
4. How were the associated decisions made?

The data was then analyzed to find dimensions, classify and assign values to these, in order to draw qualitative conclusions from the data (Patton, 2001). The significant dimensions that emerged were the phases of the resulting process, and collected data was categorized and tabulated accordingly. Contributing factors to the success or failure of the change initiatives were then identified, in order to provide the guidelines for the suggested process.

To avoid validity threats to *description*, *interpretation*, and *theory* (Robson, 2002), the interviews were recorded and transcribed, and the respondents gave feedback on this initial data. Premature hypotheses were avoided, and the respondents have given feedback on the final conclusions, along with practitioners in similar situations, described in Section 6. Section 6 also introduces a reference case which contrasts well to the results of this study.

3. Case Study Descriptions

The three companies involved in this study were industrial partners of the Center for Applied Software Engineering at Lund University (LUCAS). Part of LUCAS is the LUCAS Architecture Academy that is a one-year part time software architecture course for LUCAS partners. This research is done based on issues that came up in the context of the course. The LUCAS Architecture Academy included a number of sessions where the companies presented their architectural work to each other, and issues in the area were raised and elaborated. The companies themselves, Companies D, C and B, are presented in Chapter III. This section describes architectural changes at the companies: two at Company D, three at Company C and two at Company B. These descriptions are followed up in Section 5, with a comparison to the suggested change process.

3.1 Company D Changes

Protocol Framework: Company D recently acquired companies within their domain in order to increase their market share. The controller developed by Company D was intended to replace those companies' products. To support the same customers, the controller therefore had to support a number of legacy protocols for sensors and actuators from those products. This was realized as a problem using the present architecture, as the protocols were intertwined with the rest of the code, and could only be developed at one site, the one studied here. This site only had capacity to develop one to two new protocols per project. To be able to develop several protocols a year, Company D decided to develop a generic IO and communication protocol framework. The solution was developed through a pre-study and an implementation proposal, which resulted in a solution that enabled frequent releases of the product with many new or legacy protocols in each release. This would be accomplished by letting other departments of the company develop the protocols they were responsible for, using the protocol framework.

Real-Time Operating System: For a number of years Company D had had discussions about cutting licensing costs on Real-Time Operating Systems (RTOS). A suggestion from local product management at the studied site to develop their own RTOS was rejected by local development management. In parallel, high-level management decided to reduce the number of RTOSs to only one. This would not only lower licensing costs but also provide focus on a common competency regarding RTOSs and tool support, which would standardize and simplify distributed development. Top-level development management initiated a pre-study across all departments of the company. Participants were interviewed regarding their use of, and competencies in RTOSs. The site studied here used one RTOS, but the pre-study led to a recommendation for all departments to switch to another. Eventually the recommendation became a requirement for a project at the studied site. This requirement was postponed by the local organization, while an OS expert prepared a solution with a Virtual Operating System (VOS) layer, which was introduced in a later project.

3.2 Company C Changes

Data Router: During routine reviews the system-engineering group at Company C discovered several modules handling data streams within the system in similar ways. These modules could instead use a common data router and thereby reduce memory footprint. The architecture group developed a design proposal that was approved, but no resources were provided from the project. Project management did not consider the memory savings to be large enough. Therefore, the solution was implemented by the software architecture group and integrated with a small-scale system on an isolated branch of the code. After inspection this branch was merged with the main track, and the software architecture group initiated documentation and training on the new architectural

mechanism. The solution was still not widely accepted, as most modules already had their own implementations of the same functionality.

Hardware Abstraction Layer Split: The bottom layer of the architecture had existed in previous versions of the product, but had not been formally defined, and therefore there had been no clear rules as to how to access the hardware. The hardware was also not encapsulated well enough from the software, leading to unnecessary impact in the software when the hardware changed. The developers working in the lower layers of the product realized the need for a clearer definition of these layers. They proposed a solution that meant removing hardware dependencies from the Hardware Abstraction Layer (HAL) interface, i.e. creating a logical layer on top of the previous HAL. One driving force for introducing this logical layer was that the cost for a product developed from the platform is very dependent on the hardware components used, and therefore these are often changed to provide cheaper solutions. The purpose of the logical layer is to allow such changes without expending effort in the higher layers of the software.

The solution was presented for the system-engineering group and brought to the software architecture group. When the proposed solution was established within the system-engineering and software architecture groups, project management decided to assign resources to the change. The software architecture group introduced new coding rules according to the suggestion and made changes to the architecture descriptions. At the same time, the developers in the HAL made preparatory planning for the change, before doing the actual implementation when resources were assigned and the architecture was updated.

Include-file Reorganization: The software architecture group had created a flexible structure for the source- and include-files. The design rules that enforced this structure required several files for each component, and when the number of modules grew to around 100, the configuration management system encountered performance bottlenecks. The tool support responsibles within Company C were in contact with support personnel from the tool supplier, who identified the problem as having too many files in the system. The software architecture group was assigned to create a new structure.

The flexibility provided by the original structure was only needed by a few of the about 100 modules, and these could continue to use the previous structure. The rest of the modules were given a new structure, which basically involved merging three or four source files into one file. This resulted in a three-to-one reduction of source files.

3.3 Company B

Communication Mechanism: Company B was regularly releasing new versions of their software development tool through line-oriented development, as described in Chapter VI. New requirements, especially related to new language standards affecting the development tool being developed, meant that the old architecture could not support further development. Top-level management

therefore decided to create a new product generation. Company B had recently acquired other companies, which developed software engineering tools that were to be integrated into the new product. One of the problems with the new requirements was an increase in the number of diagram editors. The old communication mechanism did not support this increase, but one of the acquired companies had recently solved that problem, using a common object model. A technical discussion led to a consensus of using the new solution, although it meant major architectural changes.

Editor Framework: The editor framework utilized to develop graphical editors was changed as well, using a more generic solution, a decision also taken by consensus in the development project. The drivers for this change were increased reuse of common editor elements, and outsourcing of development throughout the organization. Several other decisions in this change process had to be enforced by the responsible architect, as consensus could not be reached. Both these changes were introduced in the same project.

4. Process Overview

Initial analysis of the cases led to a definition of a general process of architectural change, illustrated in Figure 15. The steps of this process where more or less present in all the seven studied changes, but there was often no conscious enactment of the process.

By analyzing the cases further, problems and opportunities were identified within the different companies, some of which are presented in Section 5. From this information, combined with theories of functional and organizational change, improvements to this process were suggested – a form in which the process is described in this section. The relationship between the process and the case studies is shown in Section 5. The purpose of the improved process is to enable organizations to make the right decisions by the right people at the right time. From an employee viewpoint the process should give guidance in the decision process, both for change initiators and decision-makers. Since architectural changes have an impact on organizations, they can be compared to the organizational change process. Kotter (1996) defines an eight-stage process which describes how to prepare an organization for major change, and how to anchor the change in the organization:

1. Establishing a sense of urgency
2. Creating the guiding coalition
3. Developing a vision and strategy
4. Communicating the change vision
5. Empowering employees for broad-based action
6. Generating short-term wins
7. Consolidating gains and producing more change
8. Anchoring new approaches in the culture

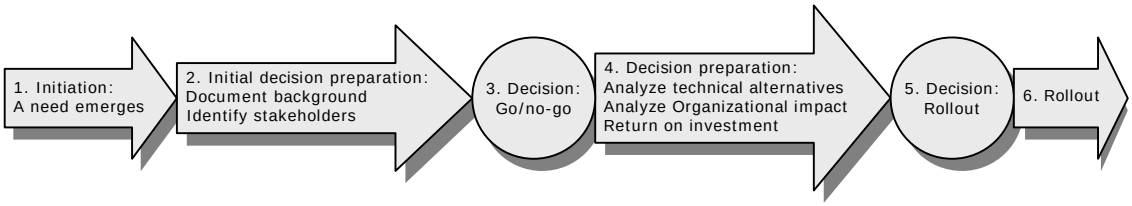


Figure 15. The Process of architectural change

The general process for functional changes involves requirements elicitation, pre-studies, implementation, and related decision-points. It focuses on how an organization should *make decisions*. Kotter's process for large-scale organizational change instead focuses on how to *make changes happen*. In the suggested process the two features are combined. This has been done by mapping Kotter's change process onto the functional change framework, such as described in the stage-gate approach (Cooper, 1990; Cooper & Kleinschmidt, 1993). In practice the process therefore has to be adapted to the present functional change framework. It must also match the balance within the organization, with respect to line and project, as discussed in Chapter VI, Figure 14. The process proposed in this paper contains the steps of Figure 15:

1. **A need emerges**: The process is preceded by a chain of events where need for change emerges or is created, and someone, the change initiator, sees this need and considers it his or her responsibility. This can be compared to Kotter's *Establishing a sense of urgency*, and to requirements elicitation in a functional change process.
2. **Initial decision preparation**: In this phase the change initiator does preparations with the goal of getting resources to analyze and implement the change.
   - **Document background**: To increase the chance of having an impact on the resolution of the need, the change initiator should document the background of the need, i.e. what products, components or organizational entities are involved, the history behind the need, how it manifests itself, etc.
   - **Identify stakeholders/decision makers**: While documenting the background, stakeholders should be identified. In order to have optimal impact, the change initiator should pay attention to these and especially to the decision makers that will be involved in the following process. This is related to Kotter's *Creating the guiding coalition*.
3. **Decision: Go/no-go**: An initial decision must be made whether the issue at hand is adequate and feasible to treat. Probably, not much effort has been spent before this decision point, e.g. one person's work for hours or days. Work done in the following phase of this process, before a decision on any particular solution or implementation of change, probably requires resources that must be budgeted, e.g. a handful of persons or more, which work for days or weeks. Therefore a person responsible for resources must

make a decision whether to go on with this process or not. The formality of this decision-point is controlled by the organization at hand. If the change can be viewed as a normal product requirement or change proposal, it can be treated as such through the ordinary channels: implementation proposal and related decision points. However, if the change is more of a change in the way people work, or a change in an internal quality attribute not leading up to completion of a specific project, the process steps that follow are of a different complexity. The risks of facing opposition are higher and the decision process and preparations must be more thorough.

4. **Decision preparation**: This phase is akin to performing a pre-study or developing an implementation proposal in technical change management. In terms of Kotter's process, it resembles *Developing a vision and strategy*.
   - **Analyze technical alternatives**: When technical alternatives have been proposed, these can be analyzed from an architectural viewpoint in a number of ways (Dobrica & Niemelä, 2002), e.g. with ATAM (Clements *et al.*, 2002).
   - **Analyze process and organization impact**: When making a technical analysis, the organizational implications are often overlooked, judging from the results of this study. This might lead to unexpected resistance to a change. An organizational analysis is therefore made, based on the initial analysis of stakeholders, in order to assess the impact of the change and prepare the organization for the change. The activity therefore contains parts of Kotter's *Communicating the change vision*.
   - **Return on investment (ROI)**: The need that the change satisfies has to have a financial side. An ROI analysis will simplify getting support for the change from top management and management of any project that might implement the change. This activity will support Kotter's *Generating short-term wins*.
5. **Decision: Rollout**: Software projects generally have a tollgate or decision point where it is decided which implementation proposals will be included in the resulting product. The same decision is made in this phase, regarding technical aspects of the architectural change. Organizational changes are however not suitable to implement in a product oriented project, and will therefore need another form of implementation and associated decision.
6. **Rollout**: This activity involves the implementation of the change. Results from this study show that the rollout of the technical part of the change often is best carried out within an ordinary project – i.e. where most organizational resources often are allocated – but other options exist. The technical implementation has to be synchronized with the rollout of the organizational change, which must be managed by, and given resources from, the line organization. This activity is related to the late phases of

Kotter's process: *Consolidating gains and producing more changes*, and *Anchoring new approaches in the culture*.

When comparing to Kotter's process it is important to keep the proper context in mind. Kotter presents a process for long-term organizational changes, which means some phases are of a different scale. Kotter's process also focuses on engaging employees and preparing an organization for a change, and not so much on how to perform the actual change. Since this paper focuses on changes to software architectures, we can use the decision framework common in software projects as a basis for a change process with features of both perspectives.

5. Analysis of Process versus Cases

This section compares the process suggested in Section 4 to the architectural changes described in Section 3. Table 13 gives an overview of the changes in relation to the suggested process, and also includes an architectural change used as a reference and presented in Section 6.

5.1 A Need Emerges

Before the suggested process is initiated a need for a change somehow appears. The reasons for changes in this study have included business decisions to increase market share, lower costs and lead time, but also more technical reasons where the architecture has not been able to support increased complexity and new features.

In Company D the need for the protocol framework was initiated when top management decided to increase the market share by acquiring other actors in the same domain. Mid-level managers and experts then saw the need for support of legacy protocols found in the newly acquired companies' products. The need for a change of RTOS on local level came from a higher-level need to save licensing costs and focus competencies by reducing the number of RTOSs. The process was initiated by higher-level management and supported by developers at other sites of the organization.

In Company C the introduction of a data router was driven by memory size being an important quality attribute. The opportunity to save memory was discovered by system engineers during routine code-reviews. The need for a HAL split emerged as the company wanted to be able to change hardware components frequently in order to save costs. The developers in the lower levels of the software initiated the change themselves in order to simplify the frequent hardware component changes. The need for an include-file restructuring became apparent, as the configuration management tool did not support the existing structure. The architecture group initiated this change since they were responsible for the include-file structure.

Two related changes were studied at Company B. A new mechanism, allowing editors to work against one system representation, was introduced to increase the number of possible editors. A framework for editor GUIs was intro-

duced in order to increase reuse of common components and enable outsourcing of editor development. Local experts initiated these changes and the technology came from the newly acquired companies. These changes would later lead to a product generation shift.

Change initiators have been identified from all levels of the companies, i.e. managers, experts appointed when the issue came up or as part of their ordinary role – where a special case is the architects themselves – and down to the developers. This can be compared to functional changes where needs often emerge from customers and are managed by marketing or product management.

5.2 Initial Decision Preparation

A decision process that can be initiated by anyone will eventually have to be brought before a decision maker. In this phase the change initiator documents the background of the issue, and identifies stakeholders and decision makers.

When the need for legacy protocol support had emerged in Company D, local experts analyzed the protocol framework solution in a pre-study. Limited attention was however paid to other departments that were supposed to implement protocols on this framework. Regarding the change of RTOS, the pre-study had been carried out by higher-level management. This resulted in recommendations to change to a single and specified RTOS. The pre-study involved interviews on all company sites.

The introduction of a data router in Company C was prepared by the system-engineering group by identifying the locations where similar functionality had been found. Current and future clients to the data router were loosely identified but not further analyzed. The only stakeholder that was approached was the architecture group, who would be responsible for developing an implementation proposal. Regarding the HAL split, the developers in that layer prepared a solution themselves, and set up a meeting with the appropriate decision-makers, in this case the system-engineering group. In the case of the include-file restructuring, the initial preparation was made by the tool-vendor's support organization. They concluded that the projects contained too many files. The architecture group was identified as a stakeholder, since they had developed the previous structure. Apart from that, stakeholder identification was not done actively, since the frequent tool failures meant that stakeholders presented themselves spontaneously.

In Company B the first steps of the product generation shift were taken on many levels, both within the original organization and by developers and managers in newly acquired organizations. Technical discussions were held, leading to the realization that the whole architecture had to be changed. Solutions were gathered from all parts of the organization, and the new architecture was adapted to enable distributed development. Stakeholders and decision-makers were therefore identified adequately.

Table 13. Overview of studied changes

	1. Initiation	2. Initial Decision Preparation	3. Decision: Go/no-go	4. Decision preparation	5. Dec: Roll-out	6. Roll-out	Organization impact
Protocol Framework	Need: Support legacy protocols – distribute protocol dev Initiator: Dev mgmt	Background: Effects of dec not considered Stakeholders: No	Dev mgmt: develop implementation proposal	Tech: Pilot study Org: Improving during process ROI: No	Postpone parts	Part of project	Large, changes of responsibility between units
RTOS switch	Need: Cut cost, standardize development Init: Top mgmt	BG: OK SH: OK	Top mgmt: req for next project	Tech: VOS on branch Org: Some ROI: No	Merge with current project	Parallel to project	Manageable changes to support contacts
Data Router	Need: Reuse to save memory Init: Sys eng	BG: OK SH: From established decision process	Sys eng: develop implementation proposal	Tech: Prestudy Org: No ROI: After the fact	No external resources	On isolated branch	Small, but developers not adopting other solutions
HAL split	Need: Simplify change of HW comp Init: Developers	BG: Development before decision SH: OK	Action already taken	Tech: See 3. Org: Low impact ROI: Obvious	Package	Part of project	New product mgmt opportunities
Include file structure	Need: Reestablish tool support Init: Arch group	BG: OK SH: OK	Internal to architecture group	Tech: 2 structures, 3 rollout alt. Org: OK ROI: Obvious	Project by project	Project by project	Change of initial process steps
Comm. Mech.	Need: Support several editors Init: Local experts	BG: Greater impact than documented SH: Some	Top mgmt: start new line of projects	Tech: Prototype project Org: No, ROI: No	Launch new series of projects	New series of projects	Change from line- to project-oriented dev
Editor framework	Need: Distribute dev Init: Local experts	BG: OK SH: OK	As above	As above	As above	As above	As above, and enabling distributed development
Ref change: Radar FW	Need: Reuse and CM Init: Developers	BG: Not explicitly SH: Not considered	No formal decision	Tech: Internal dev Org: No, ROI: No	Already implemented	Never rolled out externally	No existing org to manage new asset

In the case studies we have seen examples of less successful changes, where too little has been known about the impact of the change. Effects of functional

changes are often more limited and customer-oriented. As opposed to architectural changes, functional changes often have resources allocated to this phase, such as product management performing requirements elicitation.

5.3 Decision Point: Go/No-Go

In this activity the first decision to commit resources is made. The right decision maker shall have been defined previously, and process and organizational issues must not be forgotten in this decision.

In Company D, local level management decided that an implementation proposal of the protocol framework should be developed, since the solution would allow for more protocols and more frequent releases of the product. The organizational impact was not given much focus in this decision. Regarding the RTOS switch, top management decided to turn the recommendation into a requirement for the following projects. This requirement was later postponed by the local organization.

In Company C, the system-engineering group decided that the architecture group should develop an implementation proposal of a data router. Regarding the HAL split, the solution was so well prepared by developers that neither the system-engineering group, nor the architecture group had to invest large resources in preparation, and the decision was of little significance. Regarding the include-file structure, the architecture group themselves decided that they should develop a solution. Resources spent by the architecture group were considered insignificant in comparison to the resources wasted during tool problems. Organizational impact related to difficulties in rolling out the new structure was considered at this stage.

In Company B, the decision to apply resources to the change process was at a higher level, since it involved starting a whole new line of product-oriented projects. A decision was therefore made by top management to prepare and plan for a first project, which should result in a prototype for the product.

A forum for architecture issues could be helpful when making these decisions. Considering functional changes, organizations sometimes have product management fora, making similar decisions. The problem for the architect is that the decision is one of resources, for which the architect seldom has responsibility. Getting project resources has a benefit since the change can be more easily embraced by that project. It is however not trivial to receive resources from a project manager. The architect or change initiator therefore needs support in building a case for the proposed change.

5.4 Decision Preparation

In the decision preparation phase a small group of people will analyze technical alternatives, process and organizational impact, and ROI. From a company viewpoint this is done to make the right decision, and from an architect or change initiator viewpoint this will help convince people of the need for change. This phase is similar to developing an implementation proposal when

making a functional change, and should therefore be adapted to how implementation proposals are handled within the organization. The analysis of technical alternatives can be done in parallel with the analysis of process and organizational impact.

At Company D, the protocol framework was prepared through an implementation proposal, in the same way as a normal requirement. The technical solution was based on expert opinions. The process and organizational impact was considered, and a pilot study was made, involving development of a protocol at another site in the same company. However, there are many developers in the acquired companies that are impacted by this change but have not been involved in these initial phases. The change of RTOS was postponed to a later project, and in the meantime an OS expert prepared a solution involving a VOS layer to allow for several operating systems. One organizational impact was overlooked, as the change meant that new RTOS support contacts had to be established. Regarding ROI, the change of RTOS lead to no short-term wins for the local organization.

In Company C, the architecture group developed the data router solution in a pre-study. It was based on already implemented solutions, but the group failed to recognize opposition from project management and developers. ROI was calculated late in the process. Regarding the second change, the developers had already prepared the HAL split so the architecture group only had to prepare changes to architecture documentation and design rules. No quantitative ROI was made but the ability to change components was considered an obvious benefit, both in the short term for projects, and in the long term as a new business opportunity. Regarding the include-file restructuring, the architecture group found that the flexibility provided by the original structure was only needed in a few modules, and a simpler structure was created for other modules. ROI, or cost, was considered regarding the rollout, since rollout was expensive and did not contribute directly to any product. Short term gains were however evident, in that the current state of affairs hampered every-day operations.

In Company B the first project of the new product generation was planned. When making technical decisions many parts of the organization were involved, and consensus in joint forums was the goal. When this could not be reached, the architect responsible for that type of functionality had to make the decision. Organizational impact was not only considered when selecting solutions, but also when distributing development of various modules. This distribution could at least in one case have been better planned, as they ended up with developing a module at one site, which was highly dependent on two other modules at another site, leading to unnecessary communications problems.

The studied cases have rarely included ROI or business cases for the changes. This might be because technicians often lack knowledge of such issues, and lack insight into the business model of the company. The solution would then be either training and education, or if technically oriented change initiators would receive support from marketing and business functions. Another issue in this phase is the involvement of the right people during decision prepa-

ration. Finding the right people would in many cases aid in identifying organizational impact at an early stage.

5.5 Decision Point: Rollout

When a feature-oriented implementation proposal is completed, it is generally passed through a tollgate in the project. In this tollgate the project decides which features or implementation proposals shall be included in the upcoming release. The activity described here is similar, but the changes we have studied have had organizational impact. Such changes, and their related decisions, are hard to make in a product-oriented project.

In Company D, the implementation proposal for the protocol framework involved two different types of protocols, and a set of services for these protocols. The decision to implement was made according to the standard project model. Both protocol types were to be implemented in the upcoming project, but part of the services were postponed to later projects. Regarding the change of RTOS, the expert's VOS solution was chosen, and local development management decided to roll it out onto a current project. This project had to start implementation before the VOS was ready, and therefore local development management decided that the VOS team would make relevant modifications of the project's code when the VOS was ready to be merged with the project.

In Company C the system-engineering group approved the implementation proposal for the data router, but the architecture group did not receive project resources to implement the proposal. They then decided to implement the data router with their own resources. On the other hand, the HAL split was granted resources by project management, because it had backing from developers, system engineering, and the architecture group. The include-file restructuring was urgent, but difficult to roll out. An attempt to automate rollout failed because coding standards were not followed throughout the company. A second strategy was to halt development over a number of days, and perform the changes manually. This solution was too costly and eventually appropriate line management decided to roll the new structure out onto newly started projects, letting old projects use the old structure.

Company B decided to launch the series of projects for the new generation of products. Top management took this decision, and the content of each project has slowly been decided throughout the first projects by top management, product management and project management.

One conclusion from this activity is that it might be beneficial to restrict functional content of a new product when introducing major architectural changes. This was adequately done when introducing the IO and communication framework in Company D, as the number of services available to the protocols was restricted in the first release. Company B, however, has had problems deciding on the final content of the first product to be released on the market. Restriction of functional content is a tradeoff since customers will not accept lower functional content, and the new architecture must be able to support fu-

ture functional content. According to the theory of disruptive technology (Christensen, 1997), presented in Chapter V, it can however be the case that customers accept and want new, immature technology. Another tradeoff regarding how many future features an architecture should enable concerns the debate of programming for the future or, as XP (Beck, 1999a; 1999b) advocates, programming only for the present.

5.6 Rollout

Implementation of technical aspects of changes is made successfully within product-oriented projects. Implementing technical aspects elsewhere is more problematic, since such implementations are not so easily embraced by developers in projects. The problem is that the process and organizational aspects are often forgotten in product-oriented projects, and a standard procedure for carrying out such changes seldom exists, as opposed to carrying out a product-oriented project.

In Company D the protocol framework was implemented as part of a product-oriented project, but many departments that were intended to develop protocols have not yet had opportunity to give feedback on the framework. There is therefore still a risk that some departments will object to the framework. The VOS was developed in parallel with a product-oriented project. When the VOS was ready the two projects were merged, and the VOS team had to make remaining modifications.

In Company C the architecture group developed the data router on an isolated branch, which was later merged with the main branch. The problem was that most of the prospective clients to the new data router had already implemented their own solutions, and usage of the router was only recommended, not required. Thus it did not provide the anticipated savings in memory footprint. The HAL split had been well prepared by both developers and architects before decisions were made, and was rolled out in a project. The new include-file structure was rolled out onto one project at a time across the organization. The rollout coincided with an architectural change, which minimized overhead when the key module responsible checked in the new file structure into the configuration management system at project startup.

Company B has implemented their architectural changes in a prototype project, and a product for the market is under development. The main problems have been to settle on feature content, and as previously mentioned, the distribution of work. The company was not prepared for the organizational change they had to do in going from line-oriented development with frequent and regular releases, to project-oriented development.

The case studies include examples of changes where the technical part has been assigned to a certain project as a requirement, but postponed to later projects. There are also examples where the changes have been performed outside of product-oriented projects, further decreasing the chance of embracing the change. One of the cases made a satisfactory tradeoff, where the change of oper-

ating system was postponed to a later project, but prepared by an expert ahead of the project start.

6. Validation

In this chapter two results are presented: that there is a general process for architectural change; and that this process can be improved according to our guidelines, both in terms of value for a company, and the impact an architect can have. To study the generalizability of these qualitative results the validity has been improved in two ways.

A workshop was held with architects and system engineers from industry. Some of the companies from the initial study were present, along with some other companies, but none of the individuals were the same. One architectural change in progress was presented from each company, and it was possible to map the status of these changes onto the general change process depicted in Figure 15. However, some of the participants found it difficult to do the mapping by themselves. This still indicates that the presented process is indeed a general description of how an architecture change is performed. During the workshop, the complete process with guidelines and improvements was also presented, and the participants proposed that the concept of feedback should be introduced in the process, which is discussed in the conclusion.

In order to validate guidelines of the proposed process, a case study was made on an architectural change in a company, Company E, which develops radar systems, presented in Chapter III. This example is presented as the reference change found in Table 13, and shows when the suggested process is not applicable.

Company E had developed three similar systems in succession and developers in the later projects decided to develop a framework for the radar tracking functionality – a functionality that was similar between the three different radar sensors found in the system. Due to long project lead-time, over 5 years, the developers could implement this framework without having to rely on higher level decisions. This level of liberty however also meant that acceptance of the framework in other parts of the organization has been very low, especially from management. Although opportunities for further reuse exist, there is no funding or organization to turn the project-internal framework into an organization-wide asset. The situation resembles the HAL split in Company C, where much implementation was done ahead of any decisions to assign resources. In that case the company decision process was however eventually followed. The proposed process of architectural change can therefore be said to fit architectures that are used over time and in many products, while changes to architectures of single products developed within single projects may take place in other ways. Further evaluation is however needed in order to package and confirm the utility of the provided guidelines.

The type of systems studied here may also set bounds on the generalizability of these results. On the sliding scale from “sell and forget” products, to infor-

mation systems based on a database, these systems are closer to the packaged product end of the spectrum. The study has not included web-based systems, which often have an architecture tightly knitted to the middleware they are based on.

7. Conclusions

The case studies show that the need for architectural change can emerge from various sources, and that various roles, such as managers, architects and developers, may take responsibility for initiating change. The decisions regarding architectural changes are often carried out in the same way as companies make decisions regarding functional changes, while the implementation of architectural changes may take many forms, such as part of ordinary projects, parallel but separate projects, independent smaller projects, or as new full-scale projects.

This study reveals three major differences between functional changes and architectural changes. First of all, architectural changes are often more complex than functional changes and affect large parts of the product without showing a clear connection to a customer need. Secondly, architectural changes do not only have impact across large parts of the product, but often across the whole organization, and changes of process and organization are often overlooked and hard to implement in product-oriented projects. Finally, while companies often have mechanisms and resources in place to treat functional changes, such mechanisms are rarely established for architectural changes, and it is also hard to commit resources to activities without clear and direct customer value.

The suggested process is admittedly simplified, but must be seen as a first and feasible step for companies that do not have processes in place for architectural evolution, a situation deemed common from studying the described cases. The simplicity of the process also keeps focus on its main benefits, namely: identification of stakeholders; a defined sequence of decisions in order to gain management and organization-wide support for the initiative; and analysis of not only the technical, but also of the organizational implications of the initiative.

With increased architecture process maturity, the notion of feedback could provide further benefits from this architectural change process. In terms of the IDEAL model (McFeely, 1996), this would involve adding a 'Learning' phase to the end of the suggested process, tying back to the first phases of the process for knowledge reuse in subsequent architectural changes. In the words of the SEI Software Product Line initiative, the suggested process could then become a Software Product Line Practice Pattern (Clements & Northrop, 2001), or rather an architecture management pattern.

VIII Evolving Strategies for Software Architecture and Reuse

To achieve their business objectives, software-developing companies employ different strategies concerning architecture and reuse. These strategies include component-based development, software platforms, product lines and highly configurable code bases. Frameworks for describing these strategies have recently emerged, presenting them in orders of "increasing maturity", with researchers declaring specific architectural strategies to be more mature than others. Such frameworks can be useful for helping a company to realize a particular architectural strategy, but they do not provide guidelines concerning which architectural strategies are appropriate for companies in particular situations.

Different companies have different needs - the business context and business goals of a company will determine which architectural strategy is most suitable for that company. There is no universally "most mature" strategy. This work includes studies of the architectural situations at thirteen companies in order to determine why and how these companies have moved between architectural strategies and how these relate to reuse. The study presents a framework for describing these, and provides guidelines for companies about how to traverse the maze of architectural evolution.

From Nedstam, J. and Staples, M., "Evolving Strategies for Software Architecture and Reuse", submitted to Journal of Software Process: Improvement and Practice*, 2005.*

1. Introduction

Since the late 1960s, software reuse has been seen as one way of addressing the "software crisis" (Naur & Randell, 1968; Dijkstra, 1972). Many technical solutions for reuse have been proposed and examined, from subroutines, to modules, objects, components, and to current research into Commercial-Off-the-Shelf (COTS) components. These technical solutions have supporting tech-

nologies such as component libraries and configuration management systems. Critics of general reusable components say that reuse must be domain specific to be effective and efficient (Bosch, 2000). Reuse approaches using domain specific reusable assets are emerging, often with a strong emphasis on a common software architecture for the products reusing these assets. However, the managerial and organizational impacts of changing to various styles of reuse-oriented development have often been overlooked by researchers. To develop and maintain reusable assets, organizations need to commit resources, and need to introduce roles, team structures, and processes that plan and perform reuse-related activities. These fields of study have yet to be explored.

An architectural approach to reuse focuses efforts on the application domain at hand, i.e. the scope of the architecture. It may also support organizational adaptation, or at least put focus on issues of management and organization. A comprehensive, managed and controlled architectural approach to reuse is given in SEI's Software Product Line (SPL) framework (Clements & Northrop, 2002), presented in Chapter IV. Such an approach may be too cumbersome for small and medium sized companies to use effectively, and in response, lightweight approaches to managed reuse have been developed by e.g. Krueger (2001). These architectural approaches, as presented in Chapter IV, are prescriptive, providing a solution to managed reuse for companies with specific objectives in specific situations.

Descriptive studies have also been made, e.g. of how companies manage software product lines (Birk *et al.*, 2003), and Bosch has described a framework of levels of product line maturity (Bosch, 2002). Bosch presents guidelines as to which assets and organizational entities must be in place to successfully reach any state of the framework. The maturity states of this framework are shown in Figure 16 and are discussed in Section 5.

In summary, recent research into strategic reuse tends to advocate one strategy, such as SPL or COTS components. However, different companies have different business goals, have different resources available to meet these goals, and have to do this under different external circumstances, as in the Technology Adoption Life Cycle presented in Chapter V. The same architectural strategy will therefore not necessarily be best suited to any company in all phases of its evolution. The research goal of this study has been to provide guidelines and recommendations for companies on how to determine their most suitable architectural strategy, and how to implement this strategy.

This is a study of how software companies in quite different situations have gone about managing their reuse efforts. These studies have led to a framework for architectural evolution – a framework which extends Bosch's framework for software product line maturity. Information has been gathered from 13 companies which have performed recent architectural initiatives, and relates to how these companies have evolved along with their software architectures.

This introduction is followed by a brief method description. Section 3 gives an overview of the studied companies, Section 4 gives the dimensions by which the cases were classified, while Section 5 describes the suggested framework for

architectural evolution and how the studied companies have traversed it. Section 6 relates the observed states and transitions to the business goals of the companies involved, and the paper is concluded by a general discussion and summary.

2. Conduct of Research

The research in this study has mainly been exploratory, attempting to better understand how decisions on architectural changes are made. Information has been gathered from architectural initiatives conducted, or being under way, at 13 companies, all presented in Chapter III. Most of the 13 studied companies participated in a workgroup entitled *Platforms and Product Lines*, lead by SPIN-Syd (2005), a Swedish node of the Software Process Improvement Network. Cases from other companies have been used in order to extend and validate the results – four of these were extensions of the case studies presented in Chapter VII, while one is a research collaborator of the Empirical Software Engineering program of National ICT Australia.

The following description of the procedure of this research is summarized in Table 14. Open interviews have been the main instrument for data collection in this study. The participants of the SPIN workgroup represent eight of the 13 cases, and the focus for initial interviews was jointly discussed during three sessions in the workgroup. These interviews were to focus on current or recent architectural initiatives related to strategic reuse in the participating companies, such as the introduction of platforms for software reuse. The interviews were held to provide a description of the architectural initiative and its relation to the business model of the company, the evolution of the company and of its products. An interview guide was established for these goals. The interview guide would evolve during the course of the study.

Six participants volunteered to be interviewed, as they had recent experience of architectural initiatives. Two of these had experience from two such initia-

Table 14: Conduct of Research

Activity	Purpose
Preliminary Sessions	Establish initial research goals, identify cases, establish interview guide
Initial Interviews	Gather information, evolve interview guide
Initial Analysis	Determine and classify significant dimensions
Comparison to Previous Cases	Evolve and validate dimensions
Classification of Information	Establish dimensions and framework
Participant Feedback	Validation of facts and dimensions
Further Interview	Verification and extension of dimensions and framework
Further Analysis	Evolve framework
Peer Presentation	Validation of results

tives; hence information from eight companies was collected from the workgroup. The interviewees were in most cases originators of the architectural initiatives, but their roles in the companies varied from developers to top management. All interviews were held in an open-ended fashion (Patton, 2001), and also allowed for issues from earlier interviews to be brought up in subsequent interviews. In this way feedback from previous interviews extended the interview guide throughout the study. Most interviews were carried out one-on-one, but one interview also involved other employees from the same company. One interview included a demonstration of the product involved, while one had to be carried out over telephone. Information from another four companies was extracted from the study presented in Chapter VII, while information from an Australian company was collected in a more structured interview, following the already defined dimensions of the initial analysis. Extensive notes were taken throughout interviews, as they could not all be recorded on tape.

In order to draw qualitative conclusions, an analysis was made to extract the significant dimensions from the collected data. These were classified in a fashion such that the cases under study could be categorized along the dimensions. Information from each company was tabulated according to the categories of these dimensions. The classification of information was done in an open-ended fashion, i.e. without multiple-choice alternatives, enabling the participants to give feedback on both the facts of their companies, and on the discovered dimensions (Robson 2002, Patton 2001). The table used during the analysis is not presented in this work, but Section 4 includes the key dimensions of that table, while Table 15 gives background information on the companies, and Table 16 provides information on the dimensions related to the architectural evolution of the companies.

To minimize the threat to *description*, the participants have given feedback on these descriptions in two steps, first from the original rich transcript, and then from the material as it was presented in the resulting table. The threat to *interpretation* has foremost been mitigated by avoiding premature hypothesizing. The threat to *theory* has been handled by exposing our findings to the respondents, and to have intermediate results peer reviewed at workshops (Nedstam & Karlsson, 2004; Nedstam, 2004). A risk with the sample is that mostly initiators of the various have been sources of information for this study. Further studies will require input from other internal and external sources, but this threat has been diminished by not focusing on whether the initiatives were subjectively seen as successful or not.

3. The Studied Cases

The 13 studied companies have ranged from two to hundreds of developers, and represent suppliers of proper products, suppliers of services based on products, or of consultancy services, but also companies that are predominately customers using software components or proper products. These differences mean that some companies work in a contractual situation, while others target their

Table 15: Overview of case companies

Comp	Product	Position	Architectural Initiative	Size	Goal
B	Software development tool	Market driven product supplier	Core architecture	Medium	New markets
C	HW/SW platform for consumer electronics	Contract driven platform supplier	Packaged platform	Medium	Utilizing IP rights, creating new market
F	Consumer electronics	Market driven product supplier, buyer of platform	Acquiring packaged platform	Large	Utilizing branding opportunity
D	Control systems	Contract driven product supplier	Common base	Large	Maintain and develop many systems
G	Application platform	Contract driven platform supplier	Packaged platform	Medium	Prepare for large market
H	Publishing	Contract driven product supplier	GUI framework	Medium	Portability to 2 platforms
I	Web applications	Contract driven service provider	Application platform	Small	Streamline similar consultancy projects
J	IS development tool	Contract driven platform supplier	Application platform	Small	Kick-start for Information System development
K	Medical analysis	Contract driven product supplier	Platform for product line	Small	Expand product portfolio to similar application domain
L	Information System	Buyer of information system platform	Outsourcing application platform	Medium	Standardize a platform for a wide array of information systems
M	Combat simulator	Contract driven product supplier	Product line	Medium	Synchronize platform evolution with application projects
E	Radar	Contract driven product supplier	Sensors framework	Large	Include more types of sensors
N	Card payment	Contract driven product supplier, also sold as service	Software product line	Medium	Realize economies across product set

products for an open market. Common for all companies is that they have initiated managed reuse efforts, most often with an architectural approach. Table 15 presents a summary of the distinguishing features of the companies and of the architectural initiatives studied at the companies.

The company names have been anonymized in this study. The *Product* column indicates the domain in which each company is active, with respect to the studied architectural initiative. *Position* indicates whether the company is a buyer or supplier of a product, platform or service, in order to get a perspective on the value chain each company acts in. *Architectural Initiative* concretizes the initiative that was the subject for the particular case, and is generalized into states and transitions in Table 16. *Size* gives a rough indication of the number of developers and users affected by the initiative and *Goal* indicates the primary business objective of the initiative.

4. Dimensions from the Material

The first step of analysis was to find the main dimensions of the material. These evolved during the study. Aspects of them are shown in Tables 15 and 16, and an extract of the key dimensions from the working table are as follows:

- *Product, Service*: A description of what the company supplies to its customers, which in some cases is the software platform or architectural initiative itself. This has ranged from software development tools and information systems, to control systems and consumer electronics, but also includes services, such as those provided by either institutions utilizing a software platform or consultants.
- *Business Strategy*: This includes a company's value proposition, sales strategy and value generation. This information has been elicited in the study in order to see if and how architectural initiatives are aligned to any business strategy, and is further discussed in Section 6.
- *Architectural Initiative*: A description of the architectural initiative itself, which has ranged from GUI frameworks to introducing extensive software product lines, and packaging reusable assets as products of their own.
- *Goal and Role of Initiative*: The initiatives could be integral parts of a high-level business strategy, or could be intended to address lower-level technical goals to satisfy specific quality attributes. The initiatives studied here have covered those that have been integral parts of companies' business goals, startup companies where the architecture initiative was at the core of the business idea, and software platforms and frameworks that have emerged from technical decisions by individual developers.
- *Quality Attributes*: The "non-functional requirements" that are targeted by the architectural initiative. Most initiatives aim to lower cost or time to market through reuse, simplify change management, increase maintainability or increase understandability, but others have aimed to enable product portfolio differentiation by increasing variability and feature content.
- *Evolution*: The changes the company and the architectural initiative have gone through, and how the company has balanced its long-term and short-

term focus. Analysis of these issues is at the core of the contribution that this paper makes to the framework presented by Bosch, and is further discussed in Section 5.2.

- *Organizational Structure and Funding*: How the organization has been structured around the architectural initiative and how funding and resources have been distributed between the software platform and its products. The study includes companies that had specific organizational units for the software platform separate from the organizational units for product development, and other companies that have had quite different approaches.
- *Scope and Variation*: A description of how much of the products the platform covers, what type of domain it covers, and how the platform accommodates variation in the products. The scope has varied from specific application domain knowledge packaged in a platform for medical analysis products, to general GUI frameworks. The companies have managed variability in different ways: by using configurable product bases managed to various degrees; by performing customer projects that implement the required variability; or by focusing on one customer and after such a project refactoring the generic parts of the resulting product.

5. A Framework for Architectural Evolution

The framework of maturity levels for software product lines proposed by Bosch (2002), shown in Figure 16, was initially applied in an attempt to classify the studied cases. The original framework includes: *Independent Products,* with no reuse; *Standardized Infrastructure*, with reuse of externally developed generic assets; *Platform*, with reuse of internally developed application domain specific assets; *Software Product Line*, where variability of the platform components is managed; *Configurable Product Base*, where products are generated from the same product base; and the two composite states: *Product Population*, where less strict adherence to the product line is allowed for, and *Program of Product Lines*, where components of large systems are product lines themselves. The attempt to apply the framework to these cases led to an adaptation of the framework. By classifying the cases according to the dimensions identified in Section 4, new states and possible transitions were found in the framework. The resulting states are presented in this section, followed by the observed transitions.

5.1 States of Architectural Evolution

The "levels of maturity" for software product lines that Bosch presents are a useful framework for companies that have business goals that can be supported by a progression to product line development. However, some companies in this study had other business goals. When their evolution was compared to Bosch's framework, additional states and transitions were identified, as shown in Figure 17. In this discussion, and in Figure 17, *Product Population* is omit-

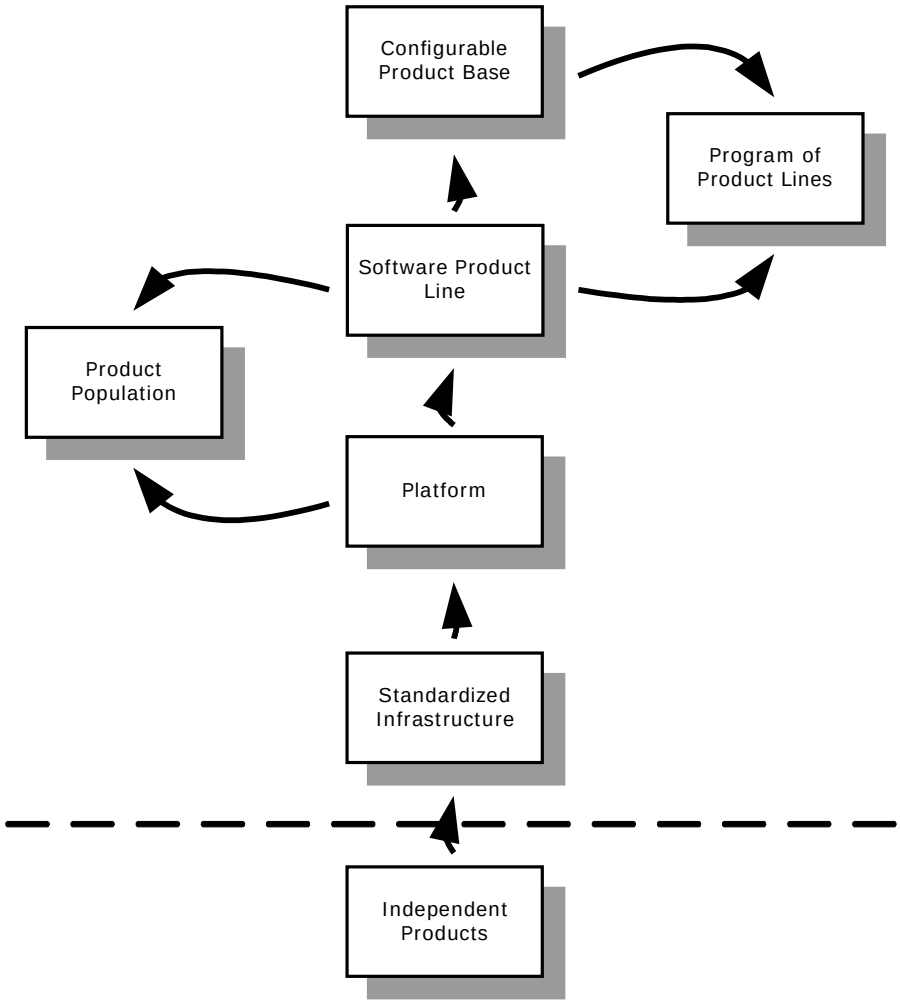


Figure 16. Maturity levels for software product lines (Bosch 2002)

ted, a state with high component development maturity but with less focus on an enforced architecture. Arrows – transitions – between states that have not been seen in the 13 cases have also been omitted. The omissions have been done to clarify the results with respect to the participating companies, and do not imply that these states and transitions are invalid. The numbers in Figure 17 refer to the enumerated transitions found in Section 5.2. States that were identified among the cases we studied were:

- **Independent Products**. In this strategy, each project developing a product stands on its own, without relying on any company-wide architectural assets.

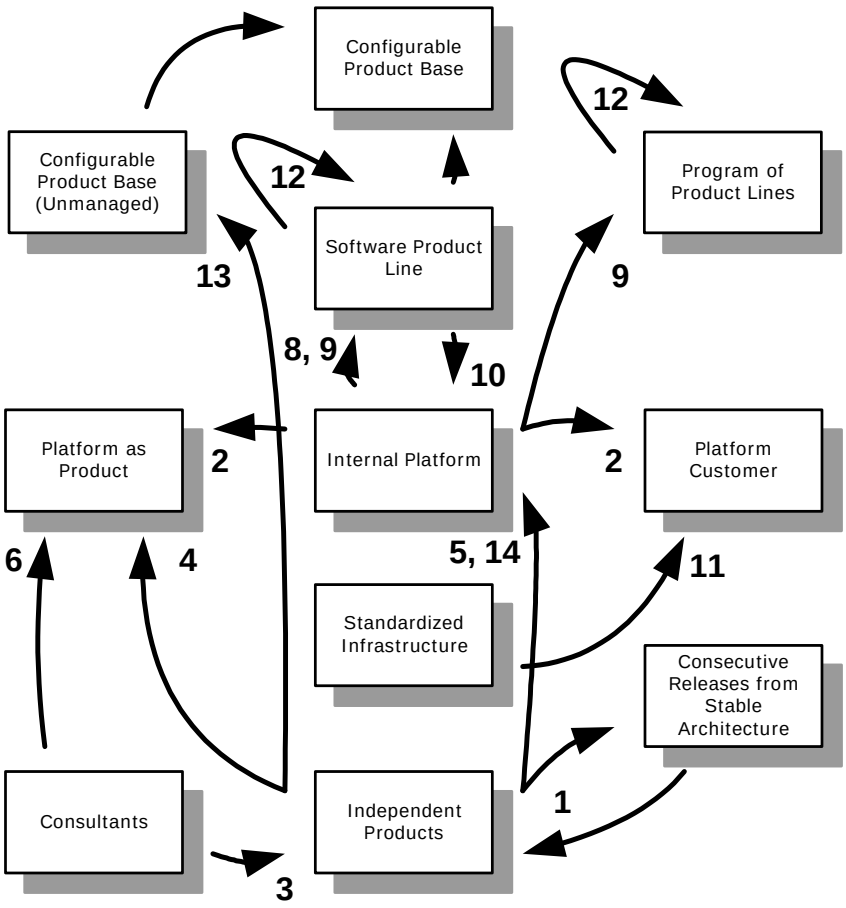


Figure 17. Transitions in architectural evolution

- **Standardized Infrastructure**. Bosch makes a distinction between a *Platform* and a *Standardized Infrastructure*, where the infrastructure is the selected operating system, database manager, or any other externally developed generic software.
- **Internal Platform**. In Bosch's original framework this state is labeled *Platform*, a description that was not adequate to describe cases we studied. We have therefore relabeled it to signify companies developing a platform intended for internal use. An *Internal Platform* is therefore developed within an organization, and is more application specific than an infrastructure component, and is reused as-is.
- **Software Product Line**. Bosch makes the distinction between *Platform* and *Software Product Line* when the platform also includes functionality that is not used by all products, leading to a need for managing variation

between components that are similar but not the same, discussed in Chapter IV. Compared to an *Internal Platform*, this state provides a complete default product, with support for introducing customer specific variation. This state was identified for companies that developed several products that are marketed simultaneously, along the lines of the software product line definitions found in Chapter IV.

- **Program of Product Lines**. This architectural strategy involves products consisting of subsystems that are themselves managed as product lines. A product is then developed by deriving its components from their respective product lines, and integrating them. Company N used this strategy when developing a system consisting of two rather independent subsystems which communicated through a small remote interface.
- **Configurable Product Base**. In this architectural strategy, products are generated or built from a common set of assets, and all products contain the same source code. Variation between products is achieved by statically or dynamically setting options provided by the assets. In Bosch's model this is considered a more mature state than *Software Product Line*.
- **Configurable Product Base (Unmanaged)**. This state is similar to *Configurable Product Base*, but companies in this state must run full-scale projects to derive a product from that product base. It is therefore also similar to *Consecutive Releases from Stable Architecture*, but companies in this category deliver and support several similar products simultaneously, not only several consecutive releases of the same product.
- **Consultants**. Our study included consultancy companies with domain expertise, some of which packaged their consultancy knowledge into products of various kinds, and this state is therefore included in the framework.
- **Consecutive Releases from Stable Architecture**. One company in particular, Company B, was able to produce new versions of their product on a regular basis, based on the previous version, but without several simultaneous variants of the product, as in the product line case. The architecture of the early versions of the product would allow such development for several releases, which warrants inclusion of this state.
- **Platform Customer**. This state could in Bosch's terminology be interpreted as *Standardized Infrastructure*, but as with *Internal Platform*, inclusion of this state is an acknowledgement of the difference between a standardized environment consisting of e.g. an operating system, database, and/or communication middleware, and a domain specific platform used for developing products within a specific application domain.
- **Platform as Product**. In this strategy, a company markets a platform intended for developing products or applications of a specific domain. These companies are the suppliers for other companies using the *Platform Customer* strategy. The study includes companies that have packaged their

platform as a product while still using it internally, companies that divested their original product development in order to focus on selling the platform as a product, and consultancy companies and startups that have packaged their knowledge in a platform to be sold as a product.

5.2 Transitions in the Architecture Lifecycle

Figure 17 also shows the transitions between states that were encountered in this work. Although the direct links from *Independent Products* to *Standardized Infrastructure* and on to *Internal Platform*, as found in Bosch's original framework, are removed from this framework, these transitions are certainly valid. The omission has been made to focus the following discussion on the observed transitions. There are no doubt other valid transitions between the presented states that have not yet been encountered, and there may well be other states that have not yet been observed.

The fact that a company can move from one state to another does not provide much information, but the reasons for such transitions do. The transitions, and their rationale in the studied cases, are presented here, numbered as in Figure 17:

1. **New product generation**: *Consecutive Releases from Stable Architecture → Independent Products*. Company B was able to develop new versions of their product in a line-oriented fashion at regular intervals for an extended period of time. The line orientation was feasible as long as individual developers or feature teams could work within well-defined modules. Eventually the architecture of the product did not support new market requirements, and the company had to develop a new product generation, i.e. go back to *Independent Products*. This was a costly move as they also had to switch over to project oriented development, which necessitates more overhead than line oriented development. This transition is not always desirable, but is sometimes necessary, and requires necessary preparations.
2. **Company split along platform interface**: *Internal Platform → Platform as Product & Platform Customer*. One company, Company A, struggling with the distribution of resources between the internal platform and the product oriented projects decided to split in two, Companies C and F, where the company that supplied the platform, Company C, was free to sell it to other customers – a novel approach to the balance between architectural assets and products. On an idealized commodity market, such a transition would reveal the inherent value of the platform.
3. **Packaging consultancy knowledge as product**: *Consultants → Independent Products*. One consultancy company, Company G, saw the opportunity of packaging their domain knowledge into a product, and could do so by an injection of venture capital. This transition is one way for consultancy companies to get revenue based on value rather than time and effort.

4. **Generalizing product into Platform as Product**: *Independent Products* → *Platform as Product*. Company G then realized that their domain knowledge was more suitable for a platform than a proper product, and a change in ownership enabled them to generalize their product into an application server. While currently living off funding from customers, they are struggling with the balance of long term and short term focus. Their current approach is to refactor their product after every customer specific project, and to avoid having separate development units concerned with their long term and short term perspective.
5. **Generalizing product into Internal Platform**: *Independent Products* → *Internal Platform*. Since the SPIN workgroup was focused on platforms, this was the most common transition. Company A did this as their first step before splitting into Companies C and F; Company H implemented a GUI framework to support multiple operating systems; and in Companies K, M and N the platform was the initial step to implement a product line. The difficulty was in all cases to get a proper level of funding for the architectural initiative. Introducing a platform in order to save costs seems to be difficult to get funding for, especially if it might mean delaying revenue from products. Company H would in hindsight have standardized an infrastructure but no suitable GUI framework COTS components were available at the time.
6. **Packaging consultancy knowledge as Platform as Product**: *Consultants* → *Platform as Product*. Company I, a consultancy company, decided to take the step directly to a platform packaged as a product. The initiative was to be done in parallel with two customer projects, funded by venture capital. It was possible to get funding as the initiative was part of a growth strategy, rather than for cost reduction.
7. **Platform as Product from startup**. A similar transition as transition 6 was performed by the startup company in the SPIN workgroup, Company J, whose first product was a platform packaged in a development tool. According to their business idea, it is not feasible for companies to fund internal platforms, but they should be packaged as products or acquired from external sources.
8. **Increased scope leading to product line**: *Internal Platform* → *Software Product Line*. Company K found opportunities to diversify their product portfolio based on their platform, leading them into a software product line. As this is a growth strategy, funding for implementation of the software product line is often easier to get.
9. **Increased platform utilization leading to product line**: *Internal Platform* → *Software Product Line* or *Program of Product Lines*. This transition is between the same states as above, but for different reasons. The companies performing this transition, Companies M and N, had about the same number of products in their portfolio before and after, and were driven to

software product line engineering in order to reduce cost or time to market when developing or maintaining products. Since this is not a growth strategy, these companies had to fund the architectural initiatives directly through customer, or product, projects. Company N transitioned to *Program of Product Lines*, because their system consisted of two distinct subsystems which were managed as separate product lines. The transition for each subsystem would be similar to a transition to the *Software Product Line* state.

10. **Decreased scope for existing product line**: *Software Product Line → Internal Platform*. Company K later reduced their scope, and might now have too extensive architectural assets.
11. **Outsource existing IT resources**: *Standardized Infrastructure → Platform Customer*. One organization, Company L, decided to reduce risk by outsourcing their IT resources, and becoming a customer rather than a developer for internal use. The strategy aims at cost reduction, but it has proven very difficult to estimate such savings. The organization has also realized that the requirements engineering capability cannot be outsourced; they still need to know what they want from their suppliers.
12. **Synchronization between applications and platform in product line setting**. This is a constant evolution in the *Software Product Line* and *Program of Product Lines* states, where development of architectural assets has to be synchronized with development of products built from these assets. This can be a cause of constant frustration for managers of these assets. Companies M and N performed this process of constant evolution of the product line and its assets. The inclusion of this transition also indicates that the states are not discrete, that companies can be in between states for lengthy periods of time while performing transitions.
13. **From contractual development to off-the-shelf products**: *Independent Products → Configurable Product Base (Unmanaged)*. Products developed under contract are often independent from each other since there is no incentive for managed reuse. The company in this situation, Company E, has a product that can be made into an off-the-shelf product, if the proper generalizations are made.
14. **Packaging project-internal platform without organizational support structure**: *Independent Products → Internal Platform*. The developers in Company E have themselves implemented a framework capturing domain knowledge, which could be the base for such generalizations, but no organization exists to manage such assets.

Table 16 gives an overview of the architectural evolution experienced by the companies, by summarizing the transitions made. The companies are again anonymized as in Chapter III, and the transitions are numbered as above. *Drivers* indicate what made the companies perform each transition, and the organizational acronyms are Domain Engineering Unit (DEU), Development De-

Table 16: Observed transitions

	Org	Tran	Drivers	Org	Solution	Scope	Variation	Quality attr
1	B	1	New market reqs	Line to project	Start from scratch	One product, new market, new standards	Family based	Time to market
2	C	5	Time to market	DEU	SW/HW platform w own resources	Specific type of consumer electronics	Demographics, geographic and standards	Time to market
3	C	2	Platform & product conflicts	DD	SW/HW platform as product	as above	as 2 and HW peripherals	Performance
4	F	5	as 2	DEU	as 2	as 2	as 2	as 2
5	F	2	as 3	BU	Platform Customer	as 2	as 2	as 2
6	D	13	Generic product, generate apps	DD	Introduce frameworks for HW	One product, which must be configured for a broad market in control systems	I/O devices, comm protocols, perf needs	Integration, performance
7	G	3	Product from domain knowledge	DD	Develop reference app	Abstract and narrow	Immature market	Functionality
8	G	4	Generalize first product	DD	Develop app server	Following standard, focused on big cust	PSTN and Internet	as above
9	H	5	Several apps, several platforms	DEU	UI framework	Mac and Windows, narrow market	Infrastructure	Cost
10	I	6	Product from domain knowledge	DEU	Tailor apps from dev wizard	E-commerce	Application servers, custom components	Cost
11	J	7	Intern platf hard to manage	DD	Platform as prod, packaged in wizard	Information systems presentation layer	Business logic implemented separately	Time to market
12	K	8	Extend scope	DEU	Extract generic parts of 1st products to SPL	2 types of medical analysis	2 domains, UI not in scope	Cost
13	K	10	Shrink scope	DEU	Platform w 1 product	1 type of medical analysis	1 domain, UI not in scope	Cost
14	L	11	Outsource development	Acquisition	Buy infrastructure	Several medical info systems	Functionality, bureaucracy	Efficiency, effectiveness
15	M	9	Customize default prod	DEU	Standardize components on platform	Tactical and strategic simulators	Several simulator views	Time to market
16	M	12	Increase SPL utilization	DEU	Generalize specific components	as above	as above	as above
17	E	14	Reuse and CM	DD	Internal HW framework	One product, increasing capability	Several sensors	Maintenance
20	N	9	Simplify platform updates	BU	File-level CM-based SPL	Banking systems	Branding, comm protocols	Cost
21	N	12	Fight reuse decay	BU	Retroactive SPL architecture	as above	as above	as above

partment (DD), and Business Units (BU), as in Bosch (2002). *Solution* again describes the architectural initiative, while *Scope* and *Variation* shows the diversity of products each architectural initiative covered. The last column indicates which quality attributes were important or affected by the transition, closely related to *Drivers*.

6. Relating Architectural Strategies to Business Goals

The study of architectural initiatives has included those implemented in response to changing business strategies, those implemented as integral parts of business strategies, and initiatives that have been more or less unrelated to any business decisions. None of these scenarios are necessarily better or worse. However, increased awareness and communication between the managerial and technical side of the companies in this study could in many cases have led to both management decisions that were more aligned to the capabilities of the development department, and developer decisions that were more targeted at the current business strategies. These issues have been affirmed by Hohmann (2003) and Faulk *et al.* (2000).

Developers have to make technical decisions to help meet their organization's business goals, and should pursue opportunities of cost reduction through reuse when this supports a business-critical objective. Senior management should on the other hand pursue opportunities of mergers, acquisitions and entering new markets, ventures that often will have technical impact. What is needed is knowledge of which strategies for architecture or reuse will best benefit the current business goals. With this knowledge, and knowledge of current business imperatives, developers would have the tools to make the correct technical decisions. Senior management would on the other hand be helped in their decision making by having a clear picture of which capabilities for reuse and architectural evolution exist when performing major changes in business models. This section discusses economic issues surrounding the various architectural strategies, and presents initial guidelines as to when each strategy is preferable.

6.1 Funding and Organizational Issues

The major management difficulty in most of the cases under study has been to find the balance between investments in reusable assets and investments in products sold to customers, i.e. the balance between long-term and short-term investments. The problem is one of funding, but also has important effects on resources, organization, and how to synchronize work.

In SEI's framework for software product line practice, 9 strategies for funding product line activities are presented:

- Product-specific funding
- Direct funding from corporate sponsor/program
- Product line organization's discretionary funds
- First product funds effort
- Multiple projects banded together to share costs
- Taxing of participating projects

- Product-side tax on customers
- Fee based on core asset usage
- Prorated cost recovery

The SEI framework also gives guidelines for their appropriateness with respect to initiating and running a software product line, and developing products from one. Bosch (2005) presents three funding strategies which are to be used with increasing maturity of software product line engineering in an organization: *bartering*, where product units developing similar components have informal structures for sharing resources to generalize such mutual assets; *taxation*, as above in the SEI framework, where projects spend part of their budget on mutual assets; and *licensing/royalties*, where a price is set on these internal assets as if they were available on the market.

Some of these funding strategies were identifiable and applicable to the studied cases, but many of the ones involving sharing costs between projects, such as taxation, were unfeasible due to cultural and political factors. The collected material also includes some interesting patterns that could extend the SEI guidelines:

- Architects often found it very easy to get funding for initiatives related to growth strategies. Cost-saving initiatives were on the other hand harder to fund – such funding was especially sensitive if the initiative would delay release of products, and therefore delay revenue. In these cases the initiatives generally had to be funded directly by customer projects. These differences could be related to which phase of the Technology Adoption Life Cycle – presented in Chapter V – a company is in.
- In young companies and startups, venture capital funding can give more liberties for long-term investments, compared to companies funded by sales and customer projects. Longer-term investment allows for visionary features and functionality, while short-term investments respond to features springing out of customer requests.
- How does a customer-specific project determine the value provided by using the architectural assets? Three approaches not conforming to the traditional product line strategy were found among the studied cases: Company A which split into Companies C and F, enabling the market to set the price of the platform; Company J that from startup decided to build a platform packaged as a product, reducing the difficulties of their customers to manage development of internal platforms; and Company L which instead decided to outsource most development. This can be seen in light of Bosch's licensing/royalties strategy (2005), which essentially is a way of packaging internal assets as products. With such a strategy one also has to be prepared to replace the internal asset with one available on the market, showing better price/performance.

Independent of funding strategy, the companies still had organizational and scheduling difficulties. Bosch presents some alternative organizational struc-

tures, three of which were represented among the studied companies. The one Bosch favors for software product line engineering is the *Domain Engineering Unit* (Bosch, 2000, 2002), where resources are dedicated to the development and maintenance of reusable assets. The other two are the *Development Department*, which develops all products and assets, and companies with a set of *Business Units*, which are responsible for one product each.

To solve the problem of synchronizing resources and scheduling between common architectural assets and product developing projects, one company split in two along the interface of the platform, so one half, Company C, now provides the platform as their product. In this way they have formalized the communication paths between their previous *Domain Engineering Unit*, and their *Business Units*. Because the platform is sold to other customers, they can also more clearly see the value of what the *Domain Engineering Unit* produces. Another company, Company J, was started with the intention of selling a platform as a product, as they consider it too hard for companies to develop internal platforms.

Company E, which is contract-driven, had no organization or resources that could package the developed framework in order for it to be reused internally. Company M, with its software product line, had a constant problem of balancing resources between reusable assets and product-specific assets; product projects would not wait for new releases of reusable assets, but rather do their own implementations of common but not yet developed functionality. One solution to this problem was to not have a separate development unit for reusable assets. Two of the companies, Companies G and J, used what Bosch calls a *Development Department* that was mainly responsible for product development, but would evolve architectural assets when necessary or between projects; they were therefore not using Bosch's suggested *Domain Engineering Unit*, to avoid synchronization problems. One company that did have a *Domain Engineering Unit*, Company H, constantly had some of its members as apprentices on the product-developing projects, to simplify requirements elicitation for the reusable assets and set the correct expectations on these assets among other developers. Whether or not a *Domain Engineering Unit* is used, the flow and prioritization of requirements from customers, over product projects, and down to the platform has been a success factor.

6.2 The Case for an Architectural Investment

A problem for initiators of architectural initiatives is to present its business case, as discussed in Chapter VII. Return on Investment is often said to be the focus when making such decisions, and this might be true if the initiator or decision maker is a product manager (Nejmeh & Thomson, 2002). The priorities of a project manager would on the other hand be to stay on time and on budget, while the role of the architect seldom is tied to any such particular responsibility. Other metrics, such as time to break-even, as discussed by SEI (Clements & Northrop, 2002), or market share, might be more suitable cornerstones of a

business case for any architecture initiative. The business goals of the company will determine how to make the business case for the architectural initiative; a business case for an architectural initiative to support growth will be very different from one to support cost reduction. None of the development level initiators in this study made a formal business case, or even produced any figures supporting the initiatives.

6.3 When to Use Each Architectural Strategy

Each architectural strategy has different impact on an organization's ability to meet different kinds of business objectives. For example, Kruger (2003) makes the case for a product line explicit, when saying that the objective of a software product line is to optimize software engineering speed and productivity by exploiting commonalities among products. However, a company with other objectives – such as high responsiveness to changing customer demands – may have to employ a different architectural strategy.

The platform strategies are suitable if some core application domain knowledge can be extracted from a set of products. A platform gets problematic when it evolves as fast as the products built from it, since one platform should sustain a number of products. If the platform and the products evolve at the same rate, managed variation is needed, which leads to a Software Product Line. The platform strategies have a narrower scope than infrastructure strategies, and the Platform as Product strategy is therefore beneficial when a company has domain specific knowledge which cannot be easily turned into a proper product, but does not have a market position where it can impose an underlying infrastructure onto a large market. Platform Customer is superior to Standardized Infrastructure if a company has a strong position in branding or application/functionality, but not so strong a position in technology.

A company having problems maintaining its Internal Platform strategy may divest either its products or its platform, and turn into a Platform Customer if the stronger position is in branding and end user functionality, or into a Platform as Product if the stronger position is in technology and intellectual property. The problems of an internal platform can be resolved by organizational measures such as rotating the developers of the platform out into the products, or by improving the requirements process from end user and customer, to products, and down to the platform itself. Managed evolution of and variation in the platform can also lead to the Software Product Line strategy.

Companies with Independent Products that see no benefits from extracting a common platform out of their products should strive to find a stable architecture for each product. When the Consecutive Releases from Stable Architecture strategy is in place, the company should be aware that this stable architecture may nonetheless eventually pose limitations on the ability to support emerging requirements or changing environments, and the specific product may have to go through a painful phase of reengineering.

Consultants with exclusive knowledge within some domain will strive to be reimbursed according to the value they generate for their customers, rather than by the hour. One way of achieving this is for them to package their knowledge in a product. Consultants may also package their knowledge in a product to be able to show prospective customers a prototype of what the consultancy could provide. If the consultancy is far away from end users, their knowledge can instead be packaged as a Platform as Product. Paradoxically, as discussed in Chapter V, product developing companies often tend to seek more stable revenues, e.g. by adding services to their products (Cusumano, 2004), services which are often charged by the hour, i.e. consultants with a quite specific domain – the product.

7. Discussion

The work shows a link between overall business goals, and the architectural strategies that best suit these goals. Companies need a *sustainable competitive advantage* (Aaker, 2001), which is based on the way the company competes (strategies, tactics), its basis for competition (resources, competencies), the market it competes in, and the competitors in that market. A strategy for sustainable competitive advantage can be derived from analysis of these factors. If this strategy is known to all parts of the organization, and is aligned to available resources, competencies and architectural assets, an architect should be able to determine the driving quality attributes for the products being developed. These quality attributes and other organizational and technical constraints may then guide the choice of architectural strategies.

In other words, the business imperatives of a company should be visible to architects and developers. From these, and the framework given here, an architect should be able to determine the most favorable state of architectural evolution, possibly with help from evaluation frameworks such as the CMM (Paulk *et al.*, 1993), ATAM (Kazman *et al.*, 1998), or the SPL Tech Probe (Clements & Northrop, 2001). The alternative strategies for funding and organization given in the framework presented here could then give the architect guidance to perform any such transition.

Establishing a set of architectural evolution state transitions, such as the one presented here, may enable us to show industry the existing options, the pros and cons of these, and in which situations each option is feasible.

7.1 Validity of Findings

While Section 2 includes a discussion of the general methods that have been applied to increase the validity of these findings, this section presents aspects of validity that are specific to the companies involved and the results produced.

Information gathered from companies in the later stages of this research has added new states to the framework, and new transitions between these states. This is a clear indication that the framework is not complete. A complete framework will however be hard to define, because competitive advantage often

can be found by finding innovative ways to compete, thereby possibly creating new states and transitions in this framework.

The set of studied companies is broad in terms of types of industry, and the generalizability in that aspect should therefore be high. It might, however, be that different industries have different patterns of evolution, and therefore should be studied separately.

The information from most of the companies in this study has been collected from initiators of the topical architectural initiatives. The bias introduced by this is hard to assess, but has been countered by not focusing on whether these initiatives have been "good" or "bad" throughout interviews and feedback sessions. Broader studies including other roles at these companies could therefore provide more confirmatory information.

The proposed model is mostly descriptive and non-predictive, so the main threat to its validity is whether the existing states and transitions are either not well enough defined to be used reliably by others, or are too specific to apply to companies that did not participate in the study. These threats are partly countered by the model's derivation from Bosch's existing product line maturity framework. Additional states in the model presented here are at a "similar" level of abstraction to Bosch's states, but address reuse strategies different to software product line development. These new states reflect commonly described practices.

7.2 Further Work

Further work needs to be done to clarify this framework, and to identify a more complete set of strategies for funding, organizing and managing architectural initiatives. Such work, based on these and previous results, should focus on describing the process of architectural evolution, in order to guide companies and individual architects to apply the benefits of an architectural approach to reuse. This would involve more analysis of the relation between business and architectural initiatives, and more analysis of ways to make business cases for different types of architectural initiatives. The framework itself can also be validated by further empirical studies, which would possibly extend the framework by identifying new states and transitions between states. The fidelity of the framework could then be examined by having experts classify a set of companies according to the framework, and measure the level of agreement between these classifications with e.g. a Kappa analysis (Fusaro *et al.*, 1997).

8. Summary and Implications

This chapter has presented qualitative analysis of information gathered from companies that have changed their architectural strategy for reuse. This analysis has informed a framework that identifies software reuse strategies, their architectural basis, and styles of architectural evolution that support organizational changes from one software reuse strategy to another.

This framework may show industry the range of options for such strategies, the pros and cons of each, and in which situations each is feasible. The framework provides a structure for the classification of organizations and their developed software. This will support further studies into areas such as alignment of business goals with architectural reuse strategies, the development of business cases for changes to such strategies, and effective team structures for such strategies.

IX Open Source Business Models in Practice: A Survey of Commercial Open Source Introduction

Open Source Software development has evolved from being an underground movement for hackers, to become a phenomenon that attracts interest from commercial organizations. The possibility to control the code, avoid vendor lock-in and accomplish cost savings makes open source software attractive. This study presents a survey of nine organizations relating to open source software in different ways. The study has resulted in documentation and analysis of the applicable business models, along with the gains and risks associated with open source software use and development. To succeed, some prerequisites need to be fulfilled. A clear strategy is needed and an organizational structure must be defined in order to assign tasks and responsibilities. There must also be a strong advocate to champion such a project and management support is crucial.

From Nedstam, J., Andersson, A. and Hässler, K., "Open Source Business Models on Practice: A Survey of Commercial Open Source Introduction", Technical Report CODEN:LUTEDX(TETS-7213)/1-14/(2005) & local 28, Department of Communication Systems, 2005.

1. Introduction

The concept of Open Source Software is fairly new in the business environment. For open source software, the source code of the software is available, which makes it possible for a software developer to understand how it is structured. The developer is permitted to make changes and to distribute the code if desirable. Open source software is usually developed in a so-called developer community, where developers from all over the world can collaborate over Internet on open source projects.

For a long time open source software was not considered to be useful outside the hacker community, but today several organizations use open source software in a commercial context. One example is Red Hat, an organization distributing Linux. Even larger enterprises such as IBM and Sun Microsystems are nowadays investing in open source software projects. Companies that are not primarily in the IT business also start to gain interest in open source software.

This work presents a survey of companies that have exploited open source software commercially. It is a study of how they have introduced open source software, which business models they have selected, but also of the risks and opportunities they have discovered in open source software.

The research question for this work has centered on analyzing open source software as part of viable business models in current industry practice. The study has investigated which types of companies have been able to profit from open source software, what the gains and risks are, the prerequisites for introducing open source software, the long-term implications of open source software, and also license and legal issues surrounding open source software. The work has been focused on open source software with licenses certified by the organization Open Source Initiative (OSI, 2005), and not on other types of free software.

This introduction is followed by a description of how the study was conducted. Section 3 introduces features of open source software that have been at issue for the survey; Section 4 describes the studied companies and how they have approached open source software; and Section 5 presents lessons learnt. Section 6 shows how the companies approached the open source community, while Section 7 discusses the business models they have utilized to gain benefits from open source software. Finally Section 8 concludes the study.

2. Conduct of Research

In this study, open-ended interviews have been used to gather information, supported by an interview guide (Robson, 2002) developed from the research questions. The participating organizations were chosen through snowball or chain sampling (Patton, 2001). In this method interviewees are asked to recommend other persons that would be valuable sources of information. For every interview, new recommendations are gathered, and the accumulated sample grows, like a snowball. After a certain time the study will converge when recommendations reoccur. When applying this method to people with open source knowledge in Swedish companies, the names of recommended persons were soon repeated. Reasons for this might be that open source is not widely spread in Swedish organizations, or that personal networks of individuals with these responsibilities are disjoint.

The interviews were carried out with two interviewers taking extensive notes, and all interviewees were given the opportunity to validate a written summary of the interview. The material was then analyzed by establishing the main dimensions of it. These were extracted from recurring themes in the inter-

views, and are presented in Section 5. The general conclusions of Section 8 have been strengthened by presenting them to, and discussing them with practitioners both skeptical and optimistic to open source software.

3. Open Source Software

This survey focuses on open source software with licenses in accordance with the Open Source Definition from the Open Source Initiative (OSI, 2005). The definition covers issues such as free redistribution, source code, derived works, and distribution of license.

The development process in open source communities is usually characterized by parallel work, independent peer review, prompt feedback, highly motivated and talented developers, increased user involvement and rapid release schedules (Feller & Fitzgerald, 2003). Eric Raymond presents the metaphor of the Cathedral and Bazaar, to contrast process-centric and open source development (Raymond, 1998). When thinking the cathedral way, bugs are assumed to be difficult to find and a few experts are assigned to this task. In the bazaar, both users and developers might search for bugs.

Open source communities do not always correspond to the picture described by Raymond. They might differ much in size and activity. Large open source projects like the development of the Linux kernel or the Apache web server are likely to profit from abundant peer review and feedback. They can also easily attract skilled developers. However, this is not necessarily true for all open source communities. Krishnamurthy (2002) presents a study of 100 mature open source software projects found on SourceForge (2005). There are few developers in most of the projects, and most projects did not generate much discussion. It is therefore not possible do draw conclusions about open source projects in general by studying e.g. the Linux kernel development.

Feller & Fitzgerald (2003) describe some of the rules in open source projects. It is e.g. important to avoid forking of a project, in order to maintain focus and a critical mass of interested developers. Another important rule is not to take credit for someone else's work. It is not common that a formal development process is used in open source projects. Tool support for version and release management is however necessary in this environment of distributed development.

In open source communities tasks are chosen voluntarily by developers, and no one can force anyone to perform a particular task (Seifert & Wieland, 2003). Therefore motivation is essential in open source software development. Bergquist & Ljungberg (2001) liken open source development to the academic way of sharing knowledge where no direct economic benefit is received from each publication. Both modes of operation also include extensive peer review.

Regarding the quality of the output of open source projects, Siefert & Wieland (2003) argue that open source projects have better quality control since developers do not have write access to the repository. Contributions from different developers are always evaluated before being integrated in the software.

Vixie (1999) believes that open source quality assurance is unorganized, but that extensive field-testing helps to improve quality.

3.1 Software Licenses

A software license generally gives a non-exclusive right to install the program on a limited number of computers (Olofsson, 2003). However, instead of limiting the user's right to the program, open source licenses increase the user's right of disposal at the author's expense.

Several open source license models exist. Some licenses are very free in the sense that they allow the user to do practically anything with the program, such as the MIT License. Others are more restrictive and impose certain requirements on the user, such as the General Public License (GPL), which require that any software derived from GPL software should be licensed under GPL (Olofsson, 2003). GPL is the most commonly used open source software license, e.g. in the Linux kernel project. GPL allows for modification of a program internally, to be used within a company for example, but if the modified program is distributed the GPL takes effect. The Lesser General Public License (LGPL) makes it possible for proprietary software development to use LGPL libraries without applying LGPL to the software.

The MIT, or X License belongs to another group of licenses along with the BSD License and the Apache Software License. These are very permissive, and only require derived source code to contain reference to the license and previous contributors.

A legal issue concerning open source software is the risk of patent infringement. It is easier to get access to the code of an open source program than a proprietary program, which makes it possible to examine the open source code searching for patent infringements. According to Golden (2004), there has however been very few instances of patent infringement during the entire history of software – a trend which might be changing as software patent regulations are changing.

4. Open Source Software Applications in Business

The open source movement has involved a great number of developers all over the world. However, when it comes to applying open source in a business context a whole new set of questions arises. In this section the case organizations' experiences of using open source are presented and discussed. Brief presentations of the companies are given in Chapter III, and an overview is given in Table 17. A number of gains and risks were revealed and are summarized in Table 18. This section is followed by a discussion of some open source business models.

Table 17. Commercial organizations using open source software.

Company	Founded in	Number of Employees	Revenues 2003	Net Income 2003
O	1966	400	$90 M	$6 M
P	1984	350	$88 M	$0.2 M
Q	1992	300	$35 M	-------
R	1911	319 273	$89.1 billion	$7.6 billion
S	1876	51 583	$20 billion	-$1.8 billion
T	1981	30	$9 M	-------
U	1999	180	-------	-------
V	1998	3 000	$480 M	-$25 M
W	1983	1 300	$295 M	-$40 M

4.1 Company O

Company O first started using software in coin sorters 15 years ago. Most machines used by customers today are based on the original software platform. This architecture has become unstructured over the years. A large mass of code made it difficult to search for bugs or make changes. To overcome these problems a new modularized platform in C++ was suggested. This work was never completed due to an overwhelming workload when writing device drivers. Instead of completing this new platform the old one was migrated from DOS to Windows. A third platform is now being developed using open source software. The ongoing project was initiated in August 2003 when the current project leader started working at Company O, championing the open source software effort. The introduction of open source software was combined with new work procedures based on Extreme Programming, XP (Beck, 1999a; 1999b). The main purpose of introducing open source software was to decrease the code base, with an ambitious target of a new code base only 10% of the size of the previous. A smaller code base would make the system easier to maintain.

A one-year period of convincing and presenting the concept preceded the introduction of open source software at Company O. The project started off by establishing an appropriate software development infrastructure. The first step was to introduce the Concurrent Versions System, CVS, instead of the former proprietary configuration management system. The extended infrastructure involves among other things a build system and issue tracking. The developed system still runs on Windows, but Linux is planned to be introduced when the organization is ready. A system built on the new platform will shortly be rolled out onto a product, and the size of the code base is 6% of the original, whereof two percent points are test cases implemented in test-driven development (Beck, 1999a; 1999b).

4.2 Company P

As early as 1997, Company P started to gain interest in Linux and open source, mainly because the servers from Sun, used at the time, were very expensive. The involvement in Linux started with developers using Linux systems for compiling. When the Linux project evolved, the possibility to run Linux on a processor without a Memory Management Unit (MMU) appeared. Company P decided to try to develop Linux for the in-house developed processor, ETRAX, and one developer was given a few months to complete the task. The project was successful and in the year 2000 the first Linux-based camera was released to the market. Since this release, all new products are developed on the Linux platform.

Within Company P, there are certain groups assigned to keep an eye on the Linux community. About ten people are responsible for updating the Linux kernel versions as well as adjusting device drivers for new versions, and another group is doing the same for applications. These groups also function as in-house support for Linux.

Two open source development projects have been initiated by Company P. In both projects, Company P first developed software in-house, and when the first version was ready, the code was released to an open source community. The first project was a Bluetooth stack, which today has been abandoned since Company P no longer focuses on the Bluetooth area. The second open source project was the Journal Flash File System (JFFS). Company P took care of the JFFS for the first two years, and then handed it over to Red Hat. These projects have contributed in making Company P well known in the world of open source.

4.3 Company Q

Company Q develops integrated hardware and software solutions for embedded real-time systems, and they started using open source software by curiosity regarding possible technical advantages. The first open source-related project at Company Q was the construction of the Wireless Open Source Platform (WOSP), which started off as a Master Thesis project. The WOSP is a system that together with a WAP-browser makes it possible to retrieve wireless information from an embedded system. It is also possible to control the embedded system from the browser. The project involved a number of different open source components that were of interest to Company Q. One of them was the open RTOS eCos, which was migrated to an ARM7-processor. Then an open Bluetooth-stack from Company P was migrated to eCos. The assignment was very successful, and led to several following projects such as the Volvo Personal Communicator (VPC) and the flight recorder StarFrec, both based on eCos.

4.4 Company R

Company R researchers started using Linux in the mid 90's, and considered it to be an additional alternative to Company R's AIX operating system. Company R has so-called Impact Teams and Business Development Teams to investigate new topics and to incubate and create new offerings to support the sales teams to bring new products to the customer. The Linux Impact team started to drive Linux and Open Source, especially on servers in 2000. Today, Company R has several thousands of clients using open source. Company R is involved in a number of open source communities, and contributes with development and bug fixing. About 300 Company R developers participate in community development. Company R has also launched the Eclipse open source project. There is a central organization within Company R that monitors activities and contributions of different open source communities. According to Company R, back-end Linux is mainstream and used by thousands of customers. However, there are not as many open source projects on the client side as on the server side today.

When a customer wants to start using open source software, a number of assessment services are offered by Company R, to investigate expectations, requirements, the skills of the employees, the infrastructure etc. Company R does not handle the open source software itself, and does not leave any warranty for it. Instead Company R focuses on services like support and installation, architecture and consolidation. Regarding development of open source software modules, Company R lets the customer decide what is to be open source, except when disclosure or nondisclosure of deliverables is part of the contract.

4.5 Company S

Company S is one of the world's leading organizations within telecom industry. Six years ago, Company S released the in-house developed programming language Erlang as open source, based on a decision not to get locked in with a technology only used by Company S and a wish for other companies to take part in the project. A couple of different products were based on Erlang at the time. Today, an open source community for Erlang exists, but no larger organization has joined the project.

Company S started to gain interest in the Eclipse project in 2002 and joined as a strategic member in 2004. There were two main reasons to start using Eclipse. Within Company S, several hundred different development tools are used and it is difficult to exchange experiences and techniques among developers. There is also much work with re-education of developers each time they join a development project, since they need to learn new development environments and languages. When using the Eclipse IDE these issues are solved, since Eclipse has the same appearance for different tools and languages. This gives a possibility to integrate tools and projects much easier.

Today, about 10-15 Company S developers work completely in Eclipse environments with development of Java based user interfaces. Two years from

now the plan is to have hundreds of Company S's 20 000 developers working in Eclipse. To make clear to suppliers that using Eclipse is part of Company S's strategy, a membership in Eclipse has seemed necessary. To be a strategic member, Company S pays an annual fee. This membership allows the company to join the board, and to make decisions about requirements, planning and architecture. Company S is involved in a number of other open source projects, and there is a group consisting of ten people working with open source and standardization

4.6 Company T

Company T has 30 employees and works mainly as a distributor of development tools for embedded systems. In recent years, there has been an increasing interest from customers regarding open source software and the possibility to save money by using open source products. This growing interest is the main reason why Company T now acts as a distributor for Company U, an organization providing an in-house developed Linux open source platform for embedded systems.

4.7 Company U

In October 2004 Company U started an open source project, with the goal of extending the Linux kernel with hard real-time abilities. IBM, Intel, Samsung, Sony, and Siemens among others support the project. Company U has earlier developed a patch to make the Linux kernel preemptible, and the 2.6 version of Linux includes this feature as standard. The goal in the project is to shorten latency times in the kernel to make them as short as in proprietary RTOSs existing today. The project is expected to finish by summer of 2005.

4.8 Company V

The Mobile Devices Smart Phone Division of Company V has in 2004 noticed an increasing interest from customers regarding Linux for smart phones. As more advanced mobile phones replace the ones used today, a different type of operating system is required. There are four main brands of operating systems on the market: Windows, Symbian, Palm and Linux. The main reason why some of Company V's customers are using or consider using Linux is the low initial price. They can simply not afford to develop mobile phones with an expensive operating system. Another motivation can be the fact that the customer is not big enough to get a source code license. The most common way to acquire Linux for mobile phones is to buy it from a distributor, usually Company U. Another, less common alternative, is to download Linux and tailor it for specific needs in-house.

4.9 Company W

Company W has been involved in, and affected by open source software in many ways – as a vendor, a developer, a competitor and a user of open source software. Company W sells tools and environments for developing open source software, but since there are open source alternatives to Company W's products, the open source movement is also a competitor. The main open source competing product is Eclipse, which is gaining market shares from JBuilder, one of Company W's most important products. Since Eclipse is a competitor, Company W has kept an eye on the Eclipse project and early became a board member. The organization is still involved in the project, but mainly as a member and does not participate with developers.

Company W has also played the role of open source developer when the database Interbase was released as open source in July 2000. The measure originated from the decision that Company W should no longer focus on this area. Instead of abandoning the project completely, the alternative of an open source project was chosen as a way to offer customers an alternative for support and development of the database. The IBPhoenix community created their database Firebird with the released code. However it became clear that some of Company W's customers preferred getting support from Company W and eventually the company reintegrated Interbase as a product of their own. Nowadays Company W's Interbase and IBPhoenix's Firebird coexist. When developing software Company W sometimes uses open source software in less business critical parts of the systems. For example, the Apache web server has been incorporated in some products.

5. Lessons Learned from Open Source Software

The following section gives an analysis of the collected material from the organizations presented above. By studying the interviews from each company, a number of positive and negative aspects of open source were found. The results are presented in Table 18, and are further discussed in Sections 5.1–5.6 where they are grouped in six different areas: Development, Security & Quality, Support, Procurement, Licenses & Legal Issues and Strategic & Organizational Aspects.

5.1 Development

When using open source, two of the studied organizations emphasized that they had managed to substantially shorten development effort, by writing less code in-house. This may cut development costs and give a better time-to-market. On the other hand it may cause developers to worry about becoming redundant and there may also be resistance to learn a new language if this is required. This was for example one of the initial problems when introducing open source software at Company O.

Table 18: Companies experiences with open source software

	Gains and Improvements	**Challenges and Risks**
Development	Possibility to view and control the code Possibility to modify the code Increased competence of own developers Knowledge exchange in the community Developers get an "ego boost" Shorter development time Cheaper development costs Easier to attract students	Strong and competent advocate needed Time to integrate OSS components Time to develop a new platform Developers may worry about becoming redundant Resistance to learn a new language OSS introduced "for fun" Staff may fear being deskilled Best developers occupied by a Linux kernel
Security & Quality	Skilled/motivated developers in the community Bugs can be fixed immediately Good testing and feedback Peer review within the community Resources for testing instead of developing Development for many OS's and environments Improved functionality	Code must be tested and verified even if it is not developed in-house*
Support	Freedom of choosing support from different providers	Need more software knowledge than when buying proprietary products Do not trust the community to solve all problems Support is not for free Professional support is not always available Need a general view of SW development*
Procurement	Get access to interesting technology No need for going through purchase department Services and software associated with open source software has in general a lower price	Spending time on downloading and evaluating open source software Do not expect OSS to be for free Lack of a specific software vendor Important to establish requirements* Important to look at all costs* Strive to use standard components*
Licenses & Legal Issues	No license cost	GPL is complicated Give attention to GPL interfaces GPL may interfere with functionality Risk loosing customers not accepting OSS Risk of patent infringement*
Strategic & Organizational Aspects	Concentrate on core competence Organizations cannot run large projects like Eclipse alone – cooperation is needed Relation between organizations in an open source software project more relaxed and non-political than otherwise An opportunity to sell more products by releasing one product for free Sell add-ons to open source software A sales argument Can create goodwill open source software is useful when a third party needs to write an application	Top management support for OSS is crucial Differ between commodities and core competences OSS project may go in other direction than wanted Critical mass of participants for OSS project A fear of failure Difficult to find short-term gains Difficult to evaluate if OSS is profitable An OSS project may be abandoned Need to incorporate components interesting to others, rather than to yourself Anyone may write an application

* Also true for proprietary software

At Company O developers became more motivated when they were introduced to community thinking and could get "ego boosts" when contributing to both in-house and community projects. Another gain was the increased compe-

tence developers achieved when working with open source. According to Company Q, there is a very efficient knowledge exchange in an open source community. An evident gain with open source software is that the code can be examined, controlled and changed as opposed to acquired proprietary software. Open source may be the only way to get access to source code, as has been the case for some of Company V's customers.

Open source is in general appealing to engineers, and open source software might sometimes be introduced as a technical challenge rather than as a good business idea. A mistake can be to involve all the best developers in configuring for example a Linux kernel, instead of developing products. It is also important to know that it takes time to integrate different open source software components. If a complete platform is to be integrated and adapted, the organization must provide resources for this. When developing a new Linux platform at Company P, two years of development work was devoted to attain the same functionality as on the previous platform. To manage all these difficulties it is important to have a strong and competent advocate leading an open source introduction. At both Company O and P management supported the idea to introduce open source software from the start.

5.2 Security and Quality

One commonly mentioned advantage of open source software is that high quality of code can be achieved. Within Company R it is believed that bug fixing and improvement is handled fast and reliably in the community. Company O argues that code of high quality can be achieved due to a large number of skilled and motivated developers in the community, and the fact that peer review is conducted frequently. Dedicated users, often developers themselves, result in a large amount of testing and feedback. An extensive developer base results in products being developed for many environments and different operating systems. Company Q points out that when you let someone else take care of development, it is possible to put more resources into testing. If bugs are found, they can be corrected immediately in-house, or by outsourcing the work.

It is important to know that the code received from the community must be tested and verified just as much as the code developed in-house, before being incorporated into products. It is also advised by Company Q that open source software should not be used in safety critical systems, such as life-supporting equipment, since it might be too unstable for these kinds of rigid requirements.

5.3 Support

When introducing open source software, it is essential for an organization to change its approach regarding software support activities. Instead of simply buying required support from the vendor providing the software, the organization must take more responsibility for support, and this is done by for example Companies P and O. Company O mainly relies on community mailing lists to get support. Company P has an in-house Linux group that is responsible for

support. Many of the organizations in the study agree that more software knowledge is needed when using open source software, compared to when buying proprietary software. Company Q suggests that someone must have a general view of the software development within the organization when using open source software. The positive side of the increased responsibility is the freedom of choosing support from different providers to fit the needs and requirements of the organization.

Company W mentions that professional support might not be available for the chosen open source software. This is something to consider when evaluating open source software products. Another important issue mentioned by Company Q, is that software problems sometimes need to be solved within the company; do not trust the open source community to solve everything. Company R does not recommend customers to use open source as a primary alternative since Company R has products of their own for which they offer support.

5.4 Procurement

To make use of open source software is, for most organizations, a way to gain access to an interesting technology. When working with proprietary software it is customary to go through a purchase department to manage software acquisition. When using open source software this is not necessary, a circumstance that gives rise to both positive and negative effects. The positive effect is that overhead expenditures can be saved if there is no need to place orders, handle deliveries etc. However, it is not advisable to let developers spend unlimited time on downloading and trying out different open source programs since these activities can be very time consuming. Company Q therefore argues the importance of determining the needed functionality. If a clear definition of wanted functionality is provided, downloading can be limited to this area. To benefit from open source software it is important to use standard software components. This is pointed out by both Company Q and O. Most open source software is developed in areas where there is a general interest.

When an organization tries to assess the consequences of introducing open source software it is important to consider all costs. Just because the software is not bought the traditional way it does not mean there are no costs to take into account. Most services and software associated with open source, such as support and open source applications, has a lower price than proprietary software. However, it is important to recognize that there might be more difficulties with open source software. Company T thinks that it is important not to expect open source software to be for free. Total Cost of Ownership (Ferrin & Plank, 2003) should therefore be investigated when considering introduction of open source software. Another factor to take into account is the relation towards an open source community, which is different from the ordinary seller-buyer relation and therefore demands a different approach. This is discussed in detail in Section 6.

5.5 Licenses and Legal Issues

The most obvious gain of introducing open source software is the fact that in most cases there are no license fees. This is mentioned by all the organizations in the study. However, a license is attached to the software and it is important to understand the conditions stated in the license, as described in Section 3.1. One of the problems observed by several organizations is the difficulty to understand GPL. It is essential to know how to treat interfaces between protected parts and GPL code. If this is not handled carefully there is a risk that code is contaminated, which means that GPL must be applied to proprietary code. Sometimes code that should have been placed in kernel space is located in user space to avoid being forced to use GPL for this code. This might deteriorate functionality.

Another difficulty concerning open source is the risk of patent infringement, as described in Section 3. Within Company R, a central office handles all legal issues, and all open source software projects must be committed by this office. According to Michel Pyschny, Leader Business Development Linux, Central Region at Company R, this office is aware of where the problems are on the market. A way to control legal issues is to understand the code of the open source software that is used. Pyschny thinks that legal issues can be a difficulty when using and developing open source, and customers might think of this as a threat. However, Pyschny does not see it as a showstopper for the open source movement. Company R is waiting for statements from the EU on software patents. Company U offers clients a warranty in case of patent infringement and takes the responsibility of any patent infringements that could be caused by the software in their distribution. One of the studied organizations lost a client that didn't want to buy products with open source software because the risk of legal conflicts with patent holders. One of Company W's clients always asks for a list of open source components. This implies a structured approach to open source software issues and to keep track of open source software programs that are accepted within the organization.

5.6 Strategic & Organizational Aspects

An important aspect to consider is what strategic direction the business should take, and how its organizational structure needs to be changed. Several of the organizations in the study believe there are both positive and negative long-term factors regarding open source software that are important to evaluate. Both Company P and Q argue the advantage of being able to concentrate on core competencies, instead of developing every component in-house. Company R relies on open source software and can therefore concentrate on business processes and consulting services. According to Company S, some development projects are too large for one company to run alone, and therefore open source development can be a solution. One example is Eclipse. Relations between organizations in these kinds of projects often seem more relaxed and non-political than in other situations where organizations interact.

It is important that management explicitly states what is to be considered as commodities and which the core competencies are. This way there will be no misunderstanding when deciding what to make open source and what to protect. According to Company R, many customers don't know what open source software means to their strategies, and studies with conflicting results confuse IT Managers. Company R's customers also find it difficult to know how to build an IT infrastructure that fits open source software.

Introducing open source software may also create goodwill for the organization, and could even be a sales argument. Company T mentions that open source is very useful when a third party wants to write an application. This is more easily done when the source code for the platform is available, and it is possible to make modifications. This could be a threat as well, since anyone may write an application to an open platform. As discussed in Section 7, there are several ways to gain revenue from open source software. Company W explains that they sell add-ons to Eclipse as a way to make money on open source software, and Company P talks about an opportunity to sell more products by releasing one product for free and being able to sell more of another product.

A large challenge for an organization on the brink of introducing open source software is the fear of failure, something Company P points out. In many organizations there is a resistance towards open source software, and a mistrust of the associated programs and development techniques. It is sometimes difficult to evaluate whether open source software is profitable, and it is usually hard to find short-term gains. Therefore, top management support is important, something Company P points out. Company R customers often introduce open source bottom-up. This is feasible in their types of projects due to Linux stability and reliability, but it would be ideal if the introduction of open source was backed up by top-down strategic decisions.

There are several risks associated with open source projects. A project can always go in another direction than the one wished for, depending on the demands from different developers and users. Company P points out that this can happen even when running an open source project of your own. It is sometimes necessary to incorporate components which are interesting to others, rather than to the owner of the project. Company Q mentions that if starting an open source software project, a critical mass of participants need to be involved, otherwise there is a risk that the project will be abandoned.

6. Relations to the Community

Besides the gains and challenges mentioned in the previous sections, a number of different ways to relate to an open source community were also revealed in the study. The most common way to relate to a community for the organizations in the study is to use open source software components in a product sold to a third party. This is done by almost all of the companies in the study. Companies P and Q point out that if something is corrected or modified in an open source software product, it is important to return the modifications to the

community; otherwise this may result in migration problems when later versions of the components are introduced. It is also important to contribute to the community in order not to be seen as a "free rider", which may result in difficulties in getting help from other developers in the project and getting your parts integrated into the project. Company P has not noticed any resistance against organizations making money on open source as long as they contribute to the community.

When using open source software, it is important to have some kind of role defined for handling the relation towards the community, which will mostly be done through mailing lists. Mailing lists can be very useful in several different ways. Companies O and P mention that it is possible to evaluate an open source project by looking at events on their mailing lists, to see how many developers that participate, and the characteristics of messages. For example, are people writing general questions like "How do I get started", or are they asking how they can move on with a certain issue? Company O explains that it is also possible to get software support and help with questions such as how license conditions should be interpreted.

Some of the companies have also released a product developed in-house as open source software, for example Companies P, W and Q. In these cases it is important that the released software is already useful to some extent, otherwise community members will not be interested in continuing the development. For the same reason, the product must be of general interest to others. Company P describes that when managing an open source project it is to some extent possible to steer it in a certain direction, but other stakeholders' opinions must also be considered. Besides running a project, it is also possible to become an influential member in a project to have impact on developed functionality, which is what Companies W and S have done when joining the Eclipse project as board members.

7. Business Models

There are many ways to define a business model. Hohensohn & Hang (2003) describe the concept as converting a business idea into a business model by orienting it and positioning it in a market to gain revenues. For a better understanding of how an organization can find a business model based on open source, the roles present on the market must be known:

- *Developers*: Private developers, professional developers and academic developers.
- *Distributors*: A distributor bundles and packages the software and offers releases in a product-style.
- *System integrators*: An integrator provides services along the value chain. The integrator offers consulting and customization of software for customers, and also integrates software into existing customer structures.
- *Software and hardware companies*: Sell software and hardware.

- *Users*: Home users, software developers, companies and institutions.

In many situations one person or company may have several of the roles above. The roles show that there are many different ways for stakeholders to interact with an open source community. Commercial organizations are interested in open source only if they can find a role that generates revenue, therefore an appropriate business model must be found.

Hohensohn & Hang (2003) describe different business models based on open source software, some of which are discussed below. The main categories are product-related and service-related business models. Some of the organizations in the study may fit into several models.

7.1 Product-Related Business Models

The product areas can be divided into operating systems, applications and appliances. Appliances are here defined as hardware combined with embedded software.

- *Operating System Distributors* add modules to the kernel, fix versions, provide documentation and package the software. They make money on deriving a product from the software. Company U with their Linux distribution fits into this model.
- *Open Source Software Application Providers* develop new applications, create releases and distribute different types of open source software or applications running on open source software. Products include applications, development tools and administrative tools. Company W sells tools for open source development and Company V has developed applications for mobile phones with the Linux operating system. Company U's Linux distribution includes tools and packages for application development.
- *Open Source Software Appliance Manufacturers* develop appliances. This is common when software itself is not creating revenues, and is provided only as a supplement to hardware. By utilizing open source products, it is possible to acquire software beyond what is possible with internal resources and knowledge. This software can also, when managed right, be achieved at low cost. This is the model that Companies O, P and Q are using for their businesses.

7.2 Service-Related Business Models

Services regarding open source software are very similar to proprietary software services.

- *Open Source Software Distribution Vendors* sell distributions together with installation services and support. Distributors of open source software usually charge a low price for their software compared to proprietary alternatives. The way to make money with this business model is instead to sell after-sale services or to sell easy to install versions. Company U provides

support and other after-sale services to the Company U Linux, and Company Q offers support services for delivered products.

- *Open Source Software Project Investors* support projects to get a larger product portfolio. Companies W, R and S are supporting the Eclipse project to obtain a certain power to inflict on the functionality developed for Eclipse. Company U sponsors a number of Linux open source projects.
- *System Integrators* offer IT services, and mainly focus on large and complex IT projects in existing IT infrastructure. Companies T, Q and V act as consultancies offering different kinds of IT-services.

8. Conclusions

The organizations in Section 4 use open source software in different ways to support their business. Some of these have open source software product-related business models, for example Company U, which develops and distributes embedded Linux. Other organizations such as Companies Q and V have models related to open source software services. Some of the organizations use open source software in embedded systems, for example Companies O and P. This shows a span of feasible business models for open source software, and a linkage to the theories on business strategy in Chapter V.

Open source software is often introduced bottom-up, since developers see many gains with using open source in their daily work. These gains include the possibility to control the code and to get assistance from other community developers. However, a bottom-up approach may not be the best way to start. For an open source software introduction to succeed it is essential to have management support and a strong advocate to lead the project. In a larger, bureaucratic organization, management approval is required before initiating open source software activities. In a small, more informal organization it might be possible for one single initiator to introduce open source software and show fast results. In both cases it is necessary to define a role concept in order to assign tasks and responsibilities. To truly profit from open source software there must be a clear strategy and the reasons for migration should be known. It must be defined which parts are core competencies and which ones are not. This definition can be applied as a guideline when deciding where open source software can be used and if in-house code can be released. It is also important to know the conditions of the open source software license in use. Some licenses are stricter than others. GPL states that all software containing, using or linking to software licensed by GPL must also be GPL. BSD is more liberal and has therefore been used in various commercial applications. Finally, the relation to the community must be clarified, such as discussions about upgrading, and how to make use of mailing lists.

There might also be long-term strategic advantages. It is for example possible to launch large development projects if several organizations cooperate, resulting in moderate costs for each organization. To make open source software a

long-term strategy it is important to monitor community activities and be prepared with alternative solutions for problems. To get respect from the community the company needs to contribute to the project in some way and not just exploit the work of others.

X Quantifying Benefits of Architecture for Selecting Components to Standardize

Traditional reuse, as well as component based and product line oriented development, relies on identification and packaging of standard components. Such efforts are often hindered by a lack of interest in long-term technology investments from the management side, and mistrust among developers regarding use of general components which might not be optimized for every particular situation. This study analyzes an attempt at discovering which components are most beneficial to standardize. The anatomy of a software system is analyzed semiautomatically, with some component attributes assigned values extracted from software development tools, and others assigned subjectively. The whole system is then optimized with regard to which components should be standardized. The validity of this approach is assessed from a measurement theory standpoint and tested on data from the case organization.

From Nedstam, J. and Höst, M., "A Quantitative Model for Valuation of Module Reusability", submitted to 10th European Conference on Software Maintenance and Reengineering, *2006.*

1. Introduction

This thesis investigates different strategies for reuse, primarily as a means to increase productivity. Reuse of standard components across organizations and domains has not yet proven successful, although the previous chapter gives example of one promising strategy. Many current efforts, such as software product lines (Clements & Northrop, 2001) have focused on domain specific reuse, which is said to be more efficient (Bosch, 2000). Low-level reuse across domains is however common, in the form of standard libraries for particular programming languages and environments. This study describes an effort to get the

best of both these worlds, an effort aiming to establish a standard library of modules within a specific domain and a specific organization.

The initiative under study is being implemented at Company F, which develops consumer electronics from a 3rd party hardware/software platform. The developed products are all belonging to the same specific application domain, and the platform is also specific to this domain. The company develops some 20 products each year – some with small variation such as internationalization, while some vary to a greater extent, such as by focusing on high-end or low-end market segments. The products contain some 150 to 200 modules, which consist of an IDL (Interface Definition Language) file and a header file, and a number of C files. Many of the modules are the same, or similar, in several products. The initiative itself involves standardizing a number of these modules which are common to all products. These standardized modules are called Foundation Modules.

The research goal of this work is to study how customized measurement models can be developed and used with credibility internally in organizations. In the situation described in this study, a model has been developed at the case company. The model is used to determine which modules should be included in the resulting set of Foundation Modules. A further challenge with this model is that it is not based on historical data, as e.g. the COCOMO model (Boehm, 1981). Since the initiative is new to the company, no such data is available, and the model is therefore based on heuristics. The research has three main objectives: to determine the validity of the model from a metrics theory standpoint (Fenton & Pfleeger, 1998); to determine if the model and its actual results are useful to the company; and to study what is required of the model to give it credibility among decision-makers within the company.

This introduction is followed by a section which describes this concept of Foundation Modules, with guidelines for using and creating them. Section 3 presents the model for selecting which modules should be turned into Foundation Modules, while Section 4 shows how this model is used by way of a small example. Section 5 presents results from running the model on an actual product, where after the study concluded.

2. The Foundation Modules

2.1 Motivation

The primary motivation for introducing this standard library of Foundation Modules is to minimize the amount of duplicated code. This will avoid inconsistencies with redundant functionality and data; reduce maintenance with reduced code size; avoid repeating updates and bug fixes to the same functionality; and lead to reduced memory footprint.

A set of Foundation Modules will also lead to increased standardization in the development environment. Different applications utilizing the library will have a similar structure. This will make it easier for developers to understand

modules they are new to, and also to introduce newly employed developers to the company.

2.2 Guidelines for the Use of Foundation Modules

As with most architecture and design initiatives, this effort will be implemented through a set of rules. Generally, a Foundation Module must be used if it provides wanted functionality, rather than implementing a new version of that functionality. The architects responsible for the Foundation Modules must also be contacted when other functionality needs to be duplicated, to determine whether that function shall be turned into a Foundation Module. These two rules are at the core of the Foundation Module concept. From a project point of view new requirements on Foundation Modules must be synchronized between projects, so that the Foundation Modules will continue to benefit all products.

2.3 Guidelines for the Implementation of Foundation Modules

The Foundation Modules' interfaces will be provided through header files and IDL files, while the modules themselves usually will be implemented in C. The interfaces should be stand-alone, i.e. not require inclusion of other include files into the header and IDL files themselves, in order to simplify use of the Foundation Modules. Internally, the Foundation Modules may only depend on other Foundation Modules, or services provided by the 3rd party platform. They must also follow the layered model which is required for the rest of the system. These measures will simplify build and linking.

Modules in general will be subject to changes that might have impact on interfaces. The company classifies interface changes according to three categories:

1. those that are breaking the compatibility of the API
2. those that are backward compatible, i.e. where a client can continue to use a new version of an API
3. and those that also are forward compatible – called API Extension, where calls can be made to dynamically determine the services provided by an API.

Compatibility breaking changes to Foundation Modules should be avoided. If deemed necessary, they must be approved by architects from the line organization, or the architect responsible for the whole project. This action will then require configuration management synchronization to introduce updated versions of the API's clients simultaneously. A backwards-compatible API change can be approved on a lower level, but will also need some synchronization since such changes require new static configurations. Finally, the architect for the individual Foundation Module can approve API extensions, and no additional configuration management or synchronization work is necessary.

This should lead to introduction of more extensible APIs, a more line-oriented development of Foundation Modules, and still allow for rapid response to new requirements on extensible Foundation Modules. The purpose is however not to make every API extensible, as there is an overhead in using such APIs. Interfaces that e.g. are not predicted to change do not need to be extensible.

3. A Model for Semiautomatic Assessment of Foundation Module Candidates

This paper analyzes an in-house model used for selecting which modules should be turned into Foundation Modules. The model gives a benefit score from a weighted average of six attributes, and is balanced by a cost factor. The model assumes that the strategic architectural decision to introduce an in-house standard library of Foundation Modules has already been made, and focuses on guiding operational decisions about which modules should be part of this standard library. Since the standard library of Foundation Modules has yet to be introduced, the model is not based on data collected on the benefits and costs involved with turning modules into Foundation Modules, but rather on heuristics on which attributes have bearing on a module's suitability as part of a standard library. Some of these attributes have to be determined by judgment calls, but the model is based on objective data which can be gathered from present software development tools. The model has been developed by establishing a clear goal of finding modules suitable for standardized reuse, with questions relating to six attributes of suitability, leading up to specific metrics to determine this suitability, along the lines of the GQM approach (Basili & Rombach, 1988).

3.1 Decision Support for Selecting Foundation Modules

The model aims at, for every module, answering the yes/no question*: Is this module suitable for turning into a Foundation Module?* This question is asked within the context of one specific product, but the future of each module is also considered, both in the context of that product, and its future in all other products developed by this company.

Each module is only a part of a system, so the answer to this question will depend on how other modules in this system behave. It will specifically depend on the modules that use and are used by this module, and whether these modules are Foundation Modules or not.

The model is used for selecting new Foundation Modules, but can also indicate that a module should no longer be a Foundation Module. If the decision has been made to turn a module into a Foundation Module, that module should be a foundation module as long as that set of products are still being developed, or at least still conform to the same system architecture. If one how-

ever would like to take a module out of the set of Foundation Modules, the cost for doing this must be analyzed through some other means.

The model focuses on the benefits of turning different modules into Foundation Modules, and can currently be used primarily to compare and rank different modules, and selecting the most beneficial as Foundation Modules. The benefit can be viewed as a percentage of the answer to the yes/no question, so a *yes, this module is suitable to turn into a Foundation Module* equals a benefit of 100%, while *no* equals 0% benefit. Cost is modeled as a fixed limit on when it actually is beneficial to turn a module into a Foundation Module. A limit of 50% would then mean that the model was very well balanced for the yes/no question above. The value for each module is therefore benefit minus limit, and a positive value indicates that the module would be beneficial to turn into a Foundation Module in the analyzed setting. The cost model is therefore very simplistic, in that each module incurs the same cost when turning it into a Foundation Module, as the limit is the same for all modules. Both cost and benefit are also relative, which makes the limit difficult to negotiate when setting parameters for the model. The roles within the organization who will make decisions based on the outcome of the model must agree on such parameters, and parameters that are intangible, hard to comprehend the meaning and effects of, or external (Fenton & Pfleeger, 1998), i.e. not possible to measure from the product itself, are harder to agree on (Lehtola & Kauppinen, 2004).

3.2 Generic Model Overview

The key idea from which the model is developed is that the dependencies between modules can be a basis for automatic assessment of the modules' suitability as Foundation Modules, as members of an in-house standard library. To each dependency there is a *required* side, which relates to how the modules supplying services to any module behave, and a *provided* side, which relates to how modules accessing services of supplier modules behave. The behavior has been classified according to the number of dependencies, and the stability of dependencies. Stability of a dependency can be affected by either changes in the interfaces between modules, or by modules being optional from product to product. Suitability could also be affected by actual functionality of a module, but such effects can not be automatically assessed, and must rather be assessed after the model has given initial decision support. The suitability measure is therefore divided into six components, as shown in Table 19.

As previously said, the benefit side of the model is a yes/no question on a floating scale. This question is therefore broken down into six sub-questions in three pairs, according to the components of Table 19. These three pairs of questions form the opinion the company has on which modules are most suitable for reuse. A company with similar objectives could use the same generic model as described here, but with their own concrete formulas, rather than the ones given in Section 4. The three pairs concern whether modules are optional or exist in all products; the way the interfaces to modules behave; and the number

Table 19. Model Components

Suitability with respect to:	Required behavior	Provided behavior
Existence stability	Yes/No	Yes/No
Interface stability	Yes/No	Yes/No
Number of dependencies	Yes/No	Yes/No

of dependencies they have. The six questions are, in a generic form: *Is module* m *a good candidate as a Foundation Module with respect to:*

Provided Existence Volatility, $\boldsymbol{S_{PEV}(m)}$*: its existence or optionality in future and present products?*

Required Existence Volatility, $\boldsymbol{S_{REV}(m)}$*: the existence or optionality of modules on which it depends?*

Provided Interface Volatility, $\boldsymbol{S_{PIV}(m)}$*: the way its interfaces are expected to maintain compatibility?*

Required Interface Volatility, $\boldsymbol{S_{RIV}(m)}$*: the way the interfaces of modules on which it depends are expected to maintain compatibility?*

Provided Dependencies, $\boldsymbol{S_{PD}(m)}$*: the number of modules that depend on this module?*

Required Dependencies, $\boldsymbol{S_{RD}(m)}$*: the number of modules this module depends on?*

The answers to these are given a value between 0 and 10, weighted and averaged. The model for the value $S(m)$ of turning one specific module m into a Foundation Module is therefore of the form:

$$S(m)=w_{PEV}S_{PEV}(m)+w_{REV}S_{REV}(m)+w_{PIV}S_{PIV}(m)+w_{RIV}S_{RIV}(m)+w_{PD}S_{PD}(m)+w_{RD}S_{RD}(m) \quad (1)$$

The total score $S(m)$ for each module is the average of these attribute scores, weighted according to w_X. The six values for $S(m)$ range from 0 to 10, and the six weights w_X should add up to 100%. The weights must also be agreed on by decision makers, and should reflect the value the organization puts on the six attributes described in the next section, with respect to a module being turned into a Foundation Module. This weighting is not trivial, as it just as the limit is difficult to illustrate and comprehend. The decision makers therefore have to understand the attributes described below. The weighting is however necessary, as an average can otherwise not be made over the six ordinal attributes (Fenton & Pfleeger, 1998).

The $S(m)$ scores model the benefit of turning a module into a Foundation Module. This benefit is balanced by S_{lim} which is a limit for what score is needed in order to get a positive answer to the question *should this module be turned into a Foundation Module*, and can also be viewed as the cost of making a module a Foundation Module. From the $S(m)$ scores a set of modules is selected to be Foundation Modules, such that the sum of differences between

score and limit over all of these Foundation Modules *S*, $\sum_{m \in FM} (S(m) - S_{\lim})$, is maximized.

The scores $S(m)$ depend on the scores of other modules in the system, specifically on whether these other modules are Foundation Modules. As the objective of the model is to select a set of Foundation Modules, this becomes an equation system with as many equations as there are modules in the system. As we shall see when the attributes are described in the following section, the individual scores increase when more modules are selected as Foundation Modules. The issue therefore becomes one of selecting modules with a high score (larger than the limit) and then experimenting with the remaining modules to see if their inclusion in the set of Foundation Modules increases the value of the others enough to outweigh the additional cost. The actual strategy for finding the best set of Foundation Modules is closely coupled to the concrete formulas in the following section, and one strategy is given in Section 5.2.

3.3 Reusability Attributes of Modules

The model currently consists of three pairs of attributes, whereby each module in the system is assigned values. These are the *required* attributes, which describe how well the suppliers of a module behave with respect to the module's ability to be a Foundation Module; and the *provided* attributes, which describe the reusability of the module itself. The three pairs of attributes are *existence volatility*, which is affected by whether the module, or its clients, is present in all products; *interface volatility*, affected by how the interface to modules change; and *dependencies*, affected by how many other modules depend on it, and how many modules it depends on itself.

Figure 18 shows the scope of the six attributes, *provided existence volatility* (PEV), *required existence volatility* (REV), *provided interface volatility* (PIV), *required interface volatility* (RIV), *provided dependencies* (PD), and *required dependencies* (RD), when analyzing the center module. This module uses the three modules below by accessing their interfaces, and provides its own interfaces to its two client modules, depicted above. This scope is further discussed towards the end of this section. These six attributes are also weighted against each other with the weights w_X of Formula 1, according to importance for any particular system or situation. The six attributes are described below.

Existence volatility concerns whether the module is present in all products that the company intends to develop. Functionality that is present in all products is more beneficiary to turn into Foundation Modules.

Provided Existence Volatility is concerning the existence of a module in all future products. A Foundation Module should be present in all future products, and the clients of such a module should be able to depend on this presence.

Required Existence Volatility is affected by the *provided existence volatility* of the modules on which a module depends. For a module to be a good Founda-

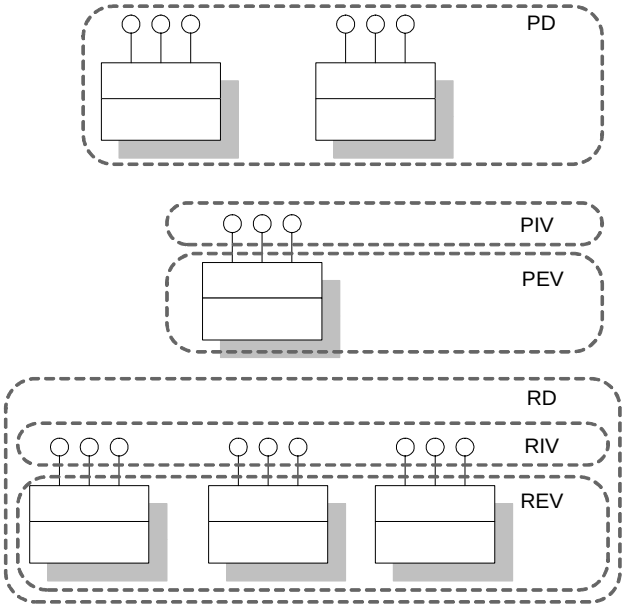


Figure 18. Scope of attributes

tion Module candidate it should not have to depend on modules that may not be present in every product.

The Foundation Modules should not only be present in all products, but they should also have stable interfaces, and rely on such interfaces.

Provided Interface Volatility concerns changes to the interfaces that the module presents to others. To be a good Foundation Module candidate, the interfaces of a module should not be subject to interface volatility. The *provided interface volatility* is judged by estimating the changes a module will be subjected to over the foreseeable future.

Required Interface Volatility concerns how the module interfaces which the module depends on will change. Foundation Modules should not have to depend on interfaces which are subject to compatibility-breaking changes.

The third pair of attributes treats the *dependencies* of modules. This is already partly covered by the previous attribute pairs, but these two attributes focus on the number of dependencies modules have.

Provided Dependencies concerns how many clients a module has. A Foundation Module can and should have many clients, as it is meant to be reused. Normal modules should not have many clients, as changes, which are more common here, then will impact more modules. The strategy of Foundation Modules is meant to bring more clients to each new Foundation Module, and therefore this relationship is less significant. However, the teams and module owners of modules that have many clients are probably used to the problems involved with adapting to requirements from several clients, and to adapting their module with minimum impact on others. A module with many clients can

also be turned into a Foundation Module to freeze it, so that it will cause less disruptive changes to its clients. Modules with several clients may therefore be more beneficial as Foundation Modules.

Required Dependencies reflects the types of modules this module depends on. As aspects of this attribute already are covered by *required existence volatility* and *required interface volatility*, see Figure 18, i.e. whether the suppliers to this module always exist and whether their interfaces change, this attribute only concerns whether the suppliers are Foundation Modules or not. The previous attributes concern the behavior of the suppliers regardless of the concept of Foundation Modules; this attribute models the change in behavior that the suppliers will exhibit if Foundation Modules are introduced. A module will be a better Foundation Module candidate if it only depends on other Foundation Modules, as these are expected to exhibit less interface changes and exist in all system configurations.

The six attributes are impacted primarily by the usage of a module in the context of a particular product and system, but assessments are also made on use of the module in other products. Existence volatility is e.g. concerned with the future of a module in all coming products. The scope of each attribute within the setting of a specific system context is shown in Figure 18. The figure shows a module under study, having two clients and three suppliers, and the scope of the six attributes. It can be seen from the figure that *required dependencies* covers the same scope as *required existence volatility* and *required interface volatility*, but as seen from the discussion above, these three concepts cover different aspects of supplier modules; especially as the former concerns the quality of these dependencies, while the latter concerns the number of these dependencies.

3.4 Determining an Optimal Set of Foundation Modules

The selection of Foundation Modules from the description of a system using the model is theoretically done by solving an equation system with as many equations as there are modules. However, the score for each module stays the same or increases when other modules – modules which this module depends on – are selected as Foundation Modules. Selection of Foundation Modules can therefore be done by choosing those that from the initial setting – with no Foundation Modules selected – that have a score larger than the cost S_{lim}. This might result in new modules which have a score larger than the cost, and the procedure is repeated. When all such modules are selected, other modules can be tried as Foundation Modules to see if they contribute more to the overall score of other modules than what they themselves bring this score down. An example of a strategy for trying out additional modules is given in section 5.2. The resulting set must then be checked to see if it complies with the original intentions of the concept of a standard library. Modules which are optional or are expected to break backwards compatibility should also be taken out of the equation initially.

4. Case Study Model Operationalization

This section gives the concrete formulas used in the model by the organization in the case study, along with discussion of their rationale and behavior. These formulas have been separated from the previous presentation as they are rich with company-specific detail and exceptions to the general description of the model. A tool for the model is presented, used with a small example system. This example system provides some concrete insights into the model behavior, while results from actual data are provided in Section 5. If these formulas are to be used in another – but similar – setting, the actual scores given to each attribute could be altered, as long as the ranks between them are maintained, and their relation to the limit S_{lim} is considered.

4.1 Existence Volatility

The score for *provided existence volatility* for any module *m*, $S_{PEV}(m)$ is

0 for modules that may not be included in all likely future products
6 for modules for which an IPR license fee is paid, and
10 for non-licensed modules that will be present in all products.

So, for the question *is this module suitable as a Foundation Module with respect to its existence in future products*, the answer is *yes* for modules that will be present in all products, *possibly* for licensed modules present in all products, and *no* for all other modules. Licensed modules are not given as high a value as internally developed modules because the company has less control over them, and therefore cannot know whether they are able to include them in all likely future products. 3rd party modules with standardized or open interfaces can however be exempt from this exception, and given the higher value.

The score for *required existence volatility* for any module *m*, $S_{REV}(m)$ is

$$S_{REV}(m) = \min(S_{PEV}(\text{each supplier to } m)) \quad (2)$$

because no Foundation Modules should depend on modules which might not be present, or

$$S_{REV}(m) = 8 \quad (3)$$

for modules which are only intended to be an isolation layer from other modules. This is not 10 as there is always drawbacks of not using e.g. a 3rd party component directly, in performance or development effort.

A module will therefore never have higher *required existence volatility* than the lowest *provided existence volatility* of any of the modules on which it depends, except for the cases where the module is an isolation layer, which gives an increase from the lower *provided existence volatility*.

The specific level of 6 is quite arbitrary, but makes sense on a ratio scale if we agree to the following statements with respect to this attribute: a licensed non-optional module is in general better than an optional module; a licensed non-optional module is more good than bad, i.e. above 5; and a module which

acts as an isolation layer is better than one which has a licensed non-optional module as client, with respect to *required existence volatility*.

4.2 Interface Volatility

Changes to module interfaces may take many forms, but are in this model classified according to three categories:

- *Compatibility-breaking changes*, where the interface no longer works for any previous clients.
- *Backward-compatible changes*, where previous clients to the module still can use the same method calls.
- *Extensible APIs*, where clients can determine the services provided by a supplier dynamically.

The use of a backward-compatible module in several products unfortunately necessitates high demands on static configurability, which means backward-compatible changes are not as simple as they appear.

The score for *provided interface volatility*, $S_{PIV}(m)$ is therefore

0 for modules where compatibility-breaking changes will occur
6 for modules where backwards-compatible changes will occur, and
10 for modules with extensible APIs.

For backwards-compatible and extensible APIs this value is decreased by 2 if the module has to be adapted for this compatibility, as an investment cost during the process of converting the candidate to a Foundation Module. Modules without client interfaces get the value 10. This means that the answer to the question *is this module a good Foundation Module candidate judging from the volatility of the interfaces it provides* is *yes* for modules with extensible APIs, *no* for modules that make compatibility-breaking changes, and *maybe* for modules that are only subject to backwards-compatible interface changes. The module is also a slightly better candidate if its behavior with respect to interface changes is already implemented.

These scores do cause some metrics theory concerns. They do give modules a higher score for better compatibility, and higher score if this implies lesser cost. But the combination of these two dimensions of this attribute gives scores that are perhaps not as intuitively evident. The score 6 is given to a module which has been prepared so that it only exhibits backwards-compatible changes, and the score 8 is given to a module which after some investment will exhibit an extensible API. This might be true, but the problem lies in that a cost is woven into this measure of benefit. This cost of preparing a module for compatibility is not the same as the cost for turning this module into a Foundation Module, but the two costs are certainly not orthogonal. A future improvement of the cost modeling could alleviate this deficiency. If the decision makers agree on the statement above, the model is still usable, and the problem merely theoretical.

The *required interface volatility* is in the standard case calculated as:

$$S_{RIV}(m) = 4 + \min(6, \min(S_{PIV}(\text{each supplier to } m))) \quad (4)$$

The exception is modules which are introduced purely to limit the effect of such changes to the rest of the system, i.e. where the module acts as an isolation layer. S_{RIV} is then set to 8. Isolation layers might be suitable either when a module is known to change interfaces often, or when several implementations of a certain functionality are available. This is in both cases especially interesting for 3rd party components, which one wants to retain the option of exchanging with other, possibly less costly, solutions.

This means that the required interface volatility varies from 4 to 10, an offset which intuitively represents a Foundation Module's ability to protect other modules from change. It could however instead be offset with the weight given to this attribute, or with the relative cost by which the Foundation Module score is compared. This discrepancy must therefore be considered when determining these weights. This attribute might get a higher impact than intended because it never reaches 0, and the weight should therefore generally be lowered, in favor of the other attributes.

The value for an isolation layer is not 10, as there is always a memory and performance overhead involved in adding a layer of indirection. It however also means that a module with suppliers which after investment only exhibit backward compatible changes is an equally good candidate as an isolation layer, and this statement must be agreed upon by the decision makers using the model.

These scores do not have the same theoretical problems as *provided interface volatility*, because both questionable scores are transformed to a contribution of 10 for this attribute. In other words: whether this module has suppliers which currently only exhibit backwards-compatible changes, or suppliers who after investment will exhibit extensible APIs, this module will still be a good Foundation Module candidate with respect to this attribute. There is however yet another theoretical problem with the exception made by isolation layers, in that the exception also affects *required existence volatility*. These two attributes are therefore not orthogonal, and the behavior of isolation layers in the model should be monitored to see what implications this has. Formula 4 also creates a tie for modules which have suppliers which will be backwards compatible, and modules which are isolation layers.

4.3 Dependencies

The *provided dependencies* score is:

$$S_{PD}(m) = 7 + \min(3, \#\text{direct clients to } m) \quad (5)$$

where the number of direct clients is determining the score. A special case is modules that will never have any clients, which get a score of 10. This means that the score for provided dependencies varies from 7 to 10. As in the case of required interface volatility, this discrepancy must be offset with the weight for

this attribute, generally speaking by lowering this weight (or that the *provided dependencies* score could be changed to range from 0 to 10). In the present setting it represents that having many clients means that a module is a better candidate, but that a module with few clients is not necessarily a bad candidate. Note that modules can get a score of 7 if they currently do not have clients, but are expected to have clients in future products.

The score for *required dependencies* is:

$$S_{RD}(m) = 10 - 2 * \min(5, \#\text{non-Foundation Module suppliers to } m) \quad (6)$$

Each additional supplier which is not a Foundation Module therefore dramatically reduces the suitability of this module as a Foundation Module. The number 5 is however somewhat arbitrary, and should be reflected in an agreement among decision makers that a module which depends on 2 non-Foundation Modules is still a rather good candidate with respect to this attribute, but a module depending on 3 such clients is a rather bad candidate.

In summary, some of the six individual attributes have theoretical deficiencies which must be considered when using the model. These deficiencies should however be feasible to explain and discuss among a group of decision makers, as should the implications of the various individual levels for the six attributes.

4.4 Extreme Attribute Behavior

Looking at extreme Foundation Module candidates, both good and bad, gives a feel for how the model behaves. The perfect Foundation Module is present in all future products, and so are the modules it depends on. It also is not expected to break any interfaces, and neither are the modules it depends on. Finally it only relies on other Foundation Modules and either has three or more clients, or is expected never to have clients. This would give a score of 10 no matter what the weights are set to.

On the other hand, modules which exhibit any of the following behavior should not be considered for Foundation Modules: optional modules; modules which are expected to break their interfaces' backwards compatibility; and modules that depend on several (5 or more) modules which are not Foundation Modules. This means that a score of 0 in any of the attributes *provided existence volatility*, *provided interface volatility* and *required dependencies*, should disqualify a module. One can therefore not only look at the total score, but must also have a multi-dimensional view of all six scores (Poladian *et al.*, 2003). This feature of the model forces decision-makers to not only consider the final score, but also consider what that score actually represents.

The total score can also not reach 0 if either the weight for *provided dependencies* or *required interface volatility* is more than 0, and can in the extreme case reach 7 out of 10. This must be considered when agreeing on S_{lim}, which generally should be set quite high.

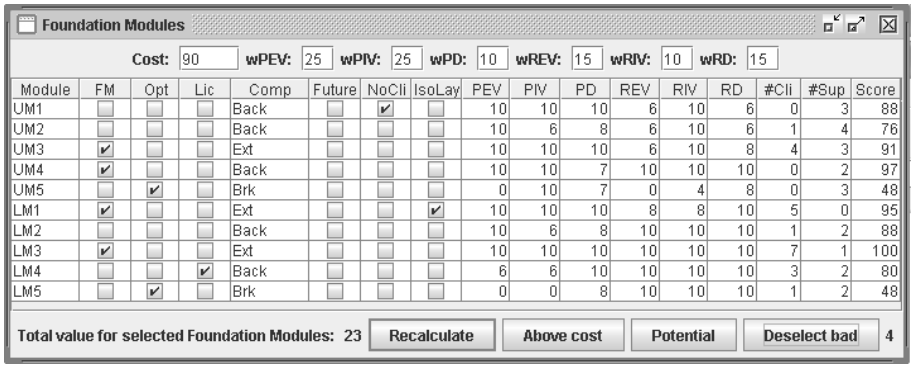


Figure 19. Foundation Module selection tool

4.5 Example System Trial

To simplify usage of the model, a tool has been developed. The tool is shown in Figure 19. It reads a dependency matrix of a set of modules, and a number of parameters for the six scores in the module: whether the module is optional; licensed; how it handles compatibility; if the compatibility is already present or will be implemented when turning the module into a Foundation Module; if the module is expected to never have clients; and if the module is an isolation layer. These six parameters can be adjusted in the tool, and the tool gives the six scores and the weighted sum for each module. When modules are selected as Foundation Modules, the sum of their scores minus S_{lim} is recalculated.

The screenshot of Figure 19 shows how the model is applied on the example system whose dependency graph is given in Figure 20. It consists of ten modules in two layers:

UM1: A shell application, which can launch other applications.
UM2: An application present in all products.
UM3: A framework for addition of registered applications.
UM4: A registered application.
UM5: An optional registered application.
LM1: A memory allocation library.
LM2: A component registration and message interchange library.
LM3: A standard library.
LM4: An inter-process communication library.
LM5: A low-level library for a registered application.

These modules have parameters according to Figure 19, which also shows a typical set of weights. In this example, modules UM5 and LM5 are disqualified because they are optional, and LM5 is also disqualified, as it is expected to break the compatibility of the interface it presents to UM5. No modules are directly disqualified because they depend on five or more non-SFMs, but the module with most suppliers, UM2, only has four suppliers. The example system

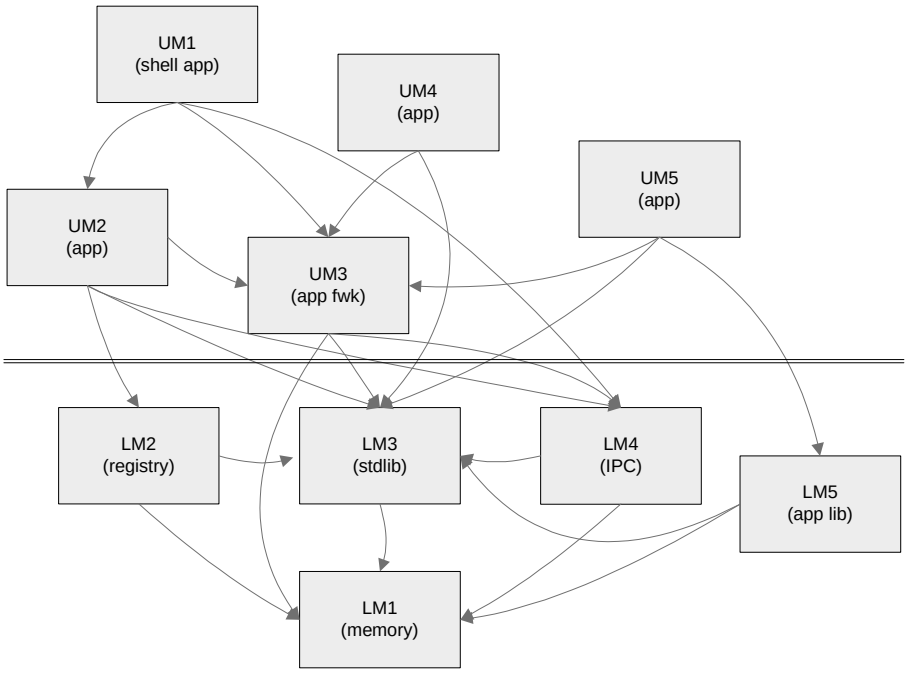


Figure 20. Example system

is therefore no optimal test of the model. It does however show something of how the model behaves.

The cost, or S_{lim}, must be set very high. Although UM5 and LM5 are disqualified, they still g*et* almost half of the possible score. This suggests that the model should first be used as a ranking of candidates, before S_{lim} is better understood among the decision makers in the company. The modules that end up being selected in the example are intuitively good candidates, perhaps with the exception of the application UM4. The other three are used by practically all other modules and act as a standard library. UM4 is on the other hand a high-level application, but gets high scores because it depends on a few stable modules and has no clients of its own. Running examples such as this shows to decision makers how the model acts, and what effect the various weights have. If an example such as this does not provide results that are according to the company's priorities, the weights or even the model itself must be changed. The implementation of the model in a tool is also a way to document the model, and such an implementation would be very familiar to a software developing company.

4.6 Model Sensitivity

The example trial also shows which parameters the model is sensitive for. As previously discussed the model has a linear sensitivity to the six weights – although the two biased scores *provided dependencies* and *required interface volatil-*

ity never reach 0. The scores themselves are however not behaving linearly to changes in the six parameters which have to be set for each module (seven with the Foundation Module parameter). The sensitivity has been assessed by changing each parameter for all modules, based on the initial setting in the example of Figure 19.

If all modules are made optional, their scores drop dramatically, although one module still reaches a total score of 70. None would however be selected as Foundation Modules in the example, from an original four. *Provided existence volatility* becomes 0 for all modules, and so does *required existence volatility* – but for the module which is an isolation layer. *Required existence volatility* also remains very low when only upper-layer modules are non-optional. If the two original optional modules are made non-optional, their score also increases dramatically, from below 50 to above 75. A change in whether a module is licensed or not has the same type of impact on the same scores, but the change is less dramatic. No licensed module will be selected at the 90 limit. If the originally licensed module is replaced with an in-house module, that module will be selected, and so will one module on which this module depends.

A change in the type of compatibility a module exhibits has similar impact, but instead on *interface volatility*. Most modules reach zero *provided interface volatility* when they are expected to break their interfaces. The *required interface volatility* does not drop as much, and the impact on the total score is therefore less dramatic than the optionality parameter. One of the example modules still receives a score above 90, but after re-selecting Foundation Modules, none of them reach the 90 limit. Setting the compatibility to "Extensible API" results in all modules receiving 10 on the *provided* score, and almost all on the *required* score. All modules except the optional ones would be selected as Foundation Modules in such a case. If all modules are to be adapted to exhibit the given compatibility, provided and required interface volatility drops marginally for some modules. This results in one of the original four Foundation Modules not reaching the 90 limit.

Changing the parameter "not expected to ever have clients" is quite theoretical for most modules, as they already have clients according to the dependency matrix. For the three modules in the example with no clients, the change is marginal when altering this value. None of them change their status as Foundation Modules. This also depends on that the weight for *provided dependencies* is only 10%, the only score affected by this parameter.

Altering whether a module is an isolation layer affects two scores, *required existence volatility* and *required interface volatility*. These scores are set to eight when a module is an isolation layer, which means that the scores can either go up or down. The *existence* score is weighted higher in the example. If the total scores are maximized with respect to this parameter, no change is required in selecting Foundation Modules. The low ranking module UM5 does however increase from 48 to 64. If isolation layers are selected to minimize the score, the

remaining modules are affected marginally, and no change in the set of selected Foundation Modules is necessitated.

If we only consider the Foundation Module selection itself, it affects the *required dependency* score positively. During selection of the four Foundation Modules in Figure 19, UM3 was not above the threshold when selection started, but gets a score above 90 when LM1 and LM3 are selected. UM2 and LM4 push UM1 above the 90 limit, but not enough to warrant inclusion of either of the pairs UM2 and UM1, or LM4 and UM1.

5. Trial on Actual Product

The model has been tried on a representative product under current development. It contains some 165 modules, and the configuration management system automatically provided the model with the dependency matrix which is the basis for the analysis. It also gave the optionality and compatibility parameters, i.e. whether modules have been present in all recent products, and the historical rate of change in interfaces. Information of whether the modules were licensed from 3rd parties was also present in the system, but difficult to extract. This information was instead provided manually by personnel at the organization. Whether a module should have improved compatibility in the future is a decision that has to be taken during the process, and no data could be generated as to whether a module is never expected to have clients, or whether a module acts as an isolation layer. The omission of these three parameters has impact on the validity of the trial performed, but as seen from the stability discussion above, these three parameters have less impact on the results than do the others. The additional effort required from the company was therefore not justifiable at the expected low improvement of this initial trial of the model.

Specifics of the data set can not be revealed, but some general features of the modules follow. 45 modules were optional, and therefore deemed impossible as Foundation Modules. 4 modules were judged to break the compatibility of their interfaces within the near future, and should likewise have been rejected had they not already been, on account of also being optional. A considerable number of modules were licensed from 3rd parties, most of which were also optional. 5 modules were deemed to have extensible APIs which would not be broken in the foreseeable future. A few modules had no clients, while one module was used by up to 158 other modules. A small number of modules had no suppliers, and the module with most dependencies had 98 suppliers.

5.1 Attribute Behavior

The *required dependencies* score behaved quite differently in the situation with 165 modules, rather than the 10 modules of the initial example. Only 13 modules had a *required dependencies* score above 0 before selection of Foundation Modules started, and while selection of one Foundation Module only has a small possible impact on each of the other modules, the impact on the total

score could sometimes be very large. This difference in behavior is expected, as the model is tailored for systems of this specific size.

One of the modules with many clients was optional, which meant that *required existence volatility* was 0 for virtually all modules. This module was also considered to break the compatibility of its interfaces in the near future, which meant that only 7 modules received a *required interface volatility* of 10, while all other modules had the minimum score of 4. In conclusion, all *required* scores are strongly negatively correlated to the number of suppliers for that module. No module with more than four suppliers received a score more than the minimum in any of these three attributes when no Foundation Modules were selected. This encompassed more than 90% of the modules.

5.2 Foundation Module Selection Strategy

The behavior of *required dependencies* requires a strategy for selecting Foundation Modules, in order not to have to try all combinations. In the case of 165 modules there are 4.7×10^{49} possible combinations, and with a million calculations a second this would take 10^{36} years. The strategy implemented in the tool is divided into three steps, where each step is iterated until it does not provide a change in the set of selected Foundation Modules before going to the next step:

1. Select all modules with a score higher than the limit. Selection of Foundation Modules increases the *required dependencies* score for client modules, and iteration provides some additional modules that get enough increase in *required dependencies* to get over the limit.
2. Select all modules, with a score less than the limit, which might provide their client modules with a potential increase in score which is larger than the drawback of selecting this sub-limit module. So if twice the number of clients who potentially may benefit from selecting this module, adjusted with the weight for *required dependencies*, is more than the difference between score and limit, the module is selected. According to the tool implementation, a client may potentially benefit if it is not yet selected, and twice the number of its non-Foundation Module suppliers, adjusted with the weight for *required dependencies*, is more than the difference between its score and the limit. This criterion could possibly be stricter, by e.g. considering that only five of these suppliers can actually play a role in increasing the score for a client. Any criterion must however result in all resulting Foundation Modules being selected, and possibly more, because the following and last step is a de-selection step. This selection step is also iterated until no more modules are selected.
3. Finally deselect modules that contribute negatively to the total system score. This way the false positives selected in the previous step are eliminated. This step must also be repeated as the *required dependencies* score of clients decrease during the de-selection procedure.

This strategy only requires around 10 iterations in the 165-module situation and can be implemented easily in a tool. The tool presently requires the user to request each iteration of the three steps manually, to better illustrate the strategy to the user. The strategy intuitively seems to select the correct set of Foundation Modules, but it has not been formally proven to do so. The strategy has been deemed good enough, considering the required effort.

5.3 Foundation Module Selection Results

Usage of the strategy on the data set, for various limits, results in a number of Foundation Modules described by Figure 21. Selecting this limit is up to the decision makers who will use the model, as is the set of six weights. In this case the same weights as in the example system of Section 4, Figure 19, have been used. Initially these decision makers must use the model to understand the effects of different settings for the limit and the weights.

Unfortunately the current settings produce a great number of ties. A limit of 63 produces a set of 8 Foundation Modules, while a limit of 62 gives 37 Foundation Modules. An appropriate set of initial Foundation Modules will probably contain between 8 and 37 modules. Of the 37, 18 are tied at a score of 63. To select a number of modules in-between 8 and 37, those that have the smallest positive impact on the total score can be taken out. In the present situation these are generally modules with few clients which score just above the limit. The model is however not to be used for automatic decision making, but rather

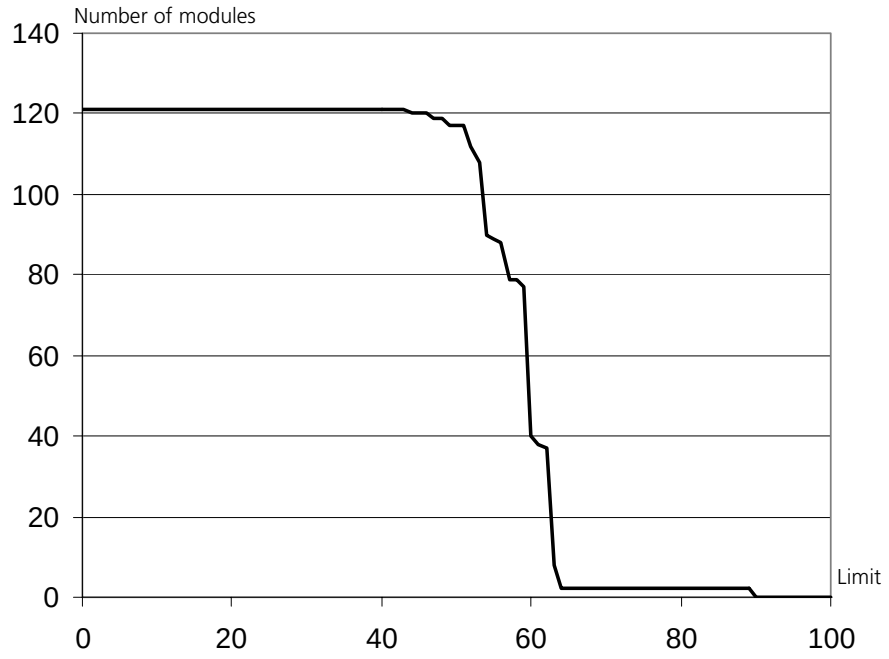


Figure 21. Number of selected modules with increasing limit

for decision support. Discussing and selecting between some 30 modules rather than some 160 modules is therefore basis for a more informed decision.

The near-step-function form of the graph could be explained by the near-step-function nature of the six attributes, and also by the positive correlation between some of these attributes – the step function behavior of any attribute will imply the same behavior in other attributes.

The model is also adapted to the current state of design quality. If design quality would be improved for all modules, the model would probably result in even more ties than at present. This would require adjustments of the model.

6. Discussion and Related Work

In the field of software engineering several cost models have been suggested, but benefit models are less frequent (Halling *et al.*, 2004). One model which attempts to quantify benefit is the CBAM (Moore *et al.*, 2003). It operates on the level of architectural strategies, where the decision to introduce the concept of Foundation Modules would be one such strategy. It would therefore be difficult to adapt to this specific situation. CBAM is based on a fixed architecture budget, while this model also tries to show – or at least currently leaves it open – how large such a budget should be. Other forms of benefit models that are emerging in the field of architecture and design are based on options theory (Asundi & Kazman, 2001; Bahsoon & Emmerich, 2003), where the flexibility provided by various solutions is quantified. One difference in approach is that the model presented here tries to quantify technical benefits from an internal perspective, rather than benefits based on external business value. As the model is used to support operational decisions, these internal benefits should be adequate and relevant. These decisions should however be checked to be in line with business decisions – e.g. that the optionality of modules is assessed from business cases, and that the rate of interface change is in line with analyses of market trends for related features.

Fenton & Pfleeger (1998) discuss two forms of reuse: reuse of existing modules found in previous projects or developed by other organizations; and internal reuse, where modules are called by several other modules within a system developed by one project. In the first reuse form, size is often the interesting factor when one wants to estimate the effort required for a project with or without reuse. The size of reused modules is measured, but also the extent to which they are changed in their new setting. The model proposed here currently does not consider the size of modules. This could be a line of further improvement, and is currently mediated in that the limit, or cost, is relative to each module.

Internal reuse is more of a complexity or structural measure, and Yin & Winchester (1978) propose one such measure:

$$r(G) = e - n + 1 \tag{7}$$

where G is a graph of n nodes and e edges describing the modules of a system, and their dependencies, akin to the dependency matrix used in the model presented here. This measure only counts the number of instances of internal reuse in a system, and quite contradicts other structural measures of tree impurity such as McCabe's cyclomatic complexity (McCabe, 1982) – an attribute which is said to have bearing on design quality (Ince & Hekmatpour, 1988). The measure sought in this work is one that shows the reusability of, and the reuse of, individual modules in a system. A measure by module analogous to that of Yin & Winchester would then only count the number of clients to a module – a measure which is only one of the six dimensions of the proposed model.

6.1 A Standard Library Comparison

Section 2 has described the rationale and design decisions behind this standard library of Foundation Modules. It is therefore inviting to compare to the design of other standard libraries. The rationale and constraints of the C++ Standard Library are described by Stroustrup (1997). When describing the scope of the library Stroustrup says "A standard library is something that every implementer must supply so that every programmer can rely on it". The library has a number of roles: to provide language features such as memory management; to handle implementation-defined language aspects; provide functions that are not optimally implemented in the language itself; provide for portability of data structures; provide a framework for extending the library in line with its principles; and provide a common foundation for other libraries.

Some of these roles are not relevant in the case of an in-house standard library for developing one type of product on one specific platform: the programming language is already selected and the scope of any portability is reduced. The remaining roles do therefore not put as heavy design constraints and demands on an in-house library, but it must still be complete and efficient enough not to tempt developers to circumvent it. One aspect that should be part of the library is in-house defined types and data structures used throughout the system – modules should be able to communicate and pass parameters without implementing duplicated functionality.

The C++ standard library has been designed with the following requirements and constraints in mind (Stroustrup, 1997):

- Invaluable and affordable to every programmer.
- Used by every programmer for everything within the scope of the library.
- Efficient enough to replace hand-coded optimization alternatives.
- Policy-free or with parameterizable policies.
- Primitive, where each component has only one role.
- Convenient, efficient and safe.
- Complete within its scope.
- Integrate well with built-in types and operations.

- Type safe.
- Supportive of commonly accepted programming styles.
- Extensible for user-defined types, to be treated the same way as built-in types.

Most of these design constraints should be true for an in-house library as well. Some of the constraints can however be relaxed in such a situation, where policies for usage of certain modules and sub-systems can be governed by design rules, and the scope of accepted programming styles can be controlled by the coding standard. Efficiency can also be tuned for the platform at hand, which means that an in-house library can have a wider scope in these respects – less functionality must be left out of the library for performance concerns. The initiative under study is also not only an attempt to create a standard library, but an attempt to create low level reusable assets for a particular application domain. This means that not only every-day development problems should be included among the Foundation Modules, but also functionality which may be present in all products, which solves problems far beyond the scope of ordinary standard libraries. The Foundation Module initiative is therefore something more than a standard library in that it allows for quite specific functionality to be included. It is also something less than a standard library, in that it does not provide qualities such as platform independency.

How does the Foundation Module initiative fare in this comparison? The principles of Foundation Modules described in the sections above partially match the principles of the C++ comparison example. But as seen in Section 4, the model for selecting the contents of this library of Foundation Modules is quite mechanical. It should therefore be viewed as a tool to find candidates, but the overall view of the standard library must be maintained. High-ranking candidates should be turned down if they are outside the scope of the library, and inclusion of some high-ranking modules may mean that lower-ranking modules within the same group of functionality also should be included. The modules that are selected in practice should be compared against the principles, and the model should be reviewed so it indeed gives decision support that follows the principles.

6.2 Validity

The concrete model has been assessed from a metrics theory standpoint, as seen from the discussion in Section 4. This assessment has shown future enhancements which are possible when more is known about costs and how the model behaves. The initial model was proposed by a practitioner at the company, who also gave most of the feedback on the results of the implemented model. The first step of strengthening the validity of these initial results must therefore be to anchor it with more stakeholders within the organization.

There is a general form to the model, and in this paper it has been specified with formulas and parameters which are company specific. The model could

therefore be adapted for companies who develop other types of systems, or who have other priorities, but still have similar reuse goals as the ones that have driven the development of this model. This possible generalizability must however first be confirmed through case studies on other organizations.

6.3 Further Work

Anchoring the model in the case organization should begin by identifying key stakeholders to the model. The primary stakeholders are those who will make decisions based on data from the model. In order to convince these roles of the usability of the model, the technical assumptions which the model is based on should first be agreed on among the developers who will be using and implementing the Foundation Modules.

The cost model is simplistic, but a proper cost evaluation would require individual analysis of each module. This cost would also be difficult to compare to the benefit generated from this model, as the cost could be determined in absolute terms, while the benefit is on a ratio scale. A proper cost could on the other hand be analyzed for the set of modules which are selected by the model. The difference between cost and benefit is also a simplistic measure, and should in the future be replaced with a proper Return-on-Investment measure. To be worthwhile, a Return-on-Investment measure should be able to model also the timing of when the investment will bring stated returns and benefits, such as through Net Present Value analysis. This model is not yet mature enough for such considerations.

The sometimes large number of ties the model produces degrades the usefulness of the model. This can be explained by that several attributes are close to step functions. *Provided dependencies* varies between 7 and 10 as the number of clients for a module varies between 0 and 3, and is thereafter 10 for any large number of clients. Similarly, *required dependencies* shows a similar pattern for the number of non-Foundation Module suppliers to a module. This could be alleviated by measuring the magnitude of, rather than the number of, clients or suppliers. This could be implemented by taking the logarithm of the module count, at an appropriate base. Such a base should be chosen to reflect the spread one wants in the particular score, and the spread of values in the data set; in our case from 7 to 10 and 0 to 10 in the scores, and 0 to 158 and 0 to 94 in the number of clients and suppliers.

7. Conclusion

The paper presents a model for assessing the benefit of reuse. The model has been developed in-house at one company, but has general characteristics, and could therefore be possible to adapt to organizations with similar objectives. Since the goal of the model has been the main driver throughout its development, the definition of metrics has naturally followed an informal GQM-like process (Basili & Rombach, 1988). Publication of company-specific models, such as the one presented here, may in the future lead to general conclusions on

how to assess benefit in Software Engineering, and how to transfer measurement theory to industry.

XI Main Contribution

This chapter provides a brief discussion of the research questions of Chapter II in relation to the presented studies. The results from each research question are discussed, and the overall conclusions, related to the research goal of Chapter I, are presented. Finally, opportunities for further work are presented.

But to introduce this final chapter, let us recapitulate the chapters of this thesis. After the introduction, method and context description, the concept of software product lines was introduced. It is one of the prime examples of an architectural strategy for increasing productivity-related quality attributes through reuse. Chapter IV presents a number of approaches to software product line engineering, and discusses different ways to introduce and maintain a software product line. The following chapter describes business and financial factors that influence the selection of software engineering strategies. It foremost introduces one model to determine the position a business has on a high-tech market – the technology adoption life cycle. Of specific interest to the market for software engineering, the difference between product- and service-based business models is also discussed. The end of Chapter V introduces a set of financial concepts that can be used to evaluate software engineering investments, and determine the cost of such investments.

The remainder of the thesis has presented the proper studies performed in the course of this work. Chapter VI introduces a method for scenario-based flexibility assessment of software development processes. More interestingly, the case study for this method shows some of the linkage between software processes, software architectures, and the organizations that develop products guided by these. This linkage is enhanced by contrasting the case study with a quite different case – to show the difference between project-oriented and line-oriented product development.

Chapter VII takes somewhat the opposite approach to investigating the linkage between software processes and software architectures – by investigating the process whereby software architectures are changed. The cases used to study the architectural change process show that this is not merely a process of technical or technological change. It is also a process of organizational change, and must be treated as such in order to be successful when performing architectural change. Changes to software architectures lead not only to changes of the soft-

ware, but also to changes in how this software is developed. This notion of architectural change leads to architectural evolution – consecutive architectural changes with an overall purpose. Chapter VIII presents a number of cases of evolution of software architectures and overall development strategies. These are combined into a framework of strategic evolution, which gives guidelines on which approaches are beneficial in which situations, and how to go about implementing them. One such strategy, open source software development, is studied in-depth in Chapter IX, with information on how a number of companies have integrated this development strategy with their business models.

The final study goes back to the technical details of the architectural change process. It presents one example of how a change initiator can go about preparing ground and gaining acceptance for an architectural change. A model for determining the benefit of one architectural strategy is presented, and the validity of this model is analyzed from a metrics theory standpoint. The example shows how established software engineering knowledge can be transferred to the practice of software architecture. With these studies fresh in memory, the overall conclusions with respect to the research questions of Chapter II can be presented.

1. Contribution

The individual studies have provided results in various specific areas. The concept of process flexibility assessment is introduced, along with an assessment method. The method was based on scenarios, and general conclusions on scenario generation and interpretation has emerged. Similar conclusions have been provided while using process simulation to better understand the case studies. Light has been shed on issues such as funding of architectural initiatives; challenges and opportunities with open source development have been analyzed; and also the challenge of introducing metrics in practice. The studies as a whole have however primarily aimed at the research goal presented in Chapter II. The contribution of this work is described as a discussion of the research questions and how they relate to results found in the presented studies, and a discussion of general conclusions around the goal of this thesis.

1.1 Research Questions

Question 1: What is the linkage between architectural strategies and business goals?

The business goals of the companies studied in this thesis, with regards to the focus topics of the thesis itself – i.e. architectural strategies to improve quality attributes related to productivity – have been in the fields of cost savings, lead-time reductions, entering new markets, inventing new markets and focusing on core competencies. The business goals that have been covered in this thesis, primarily in Chapters VII and VIII, are summarized in Table 19. As can be seen from the table the quality attributes sought for by the companies in their architecture-related initiatives are close to their business goals. In general

Table 19: Overview of case companies

Comp	Qualities	Goal type	Goal
B	Extendibility, integrability	Expand	Enter new markets
C	Performance, open interfaces	Innovate	Utilizing IP rights by creating new market
F	Lead time, feature content	Expand, focus	Utilizing branding opportunity and increase diversification to increase market share
D	Extendibility, cost	Expand, cost	Increase market share through integration of technologies; save costs.
G	Standards, infrastructure	Innovate	Take business from early market of customized solutions to mature market of standard solutions
H	Portability	Expand	Portability to 2 platforms to increase market share.
I	Lead-time	Innovate	Streamline similar consultancy projects, take business from early customized market to mature standardized market.
J	Flexibility	Innovate	Capitalize on companies desires to outsource parts of Information System development
K	Extendibility	Expand, time	Enter new niche market by expanding product portfolio; shorten lead-time.
L	Cost, interoperability	Integrate, cost	Standardize a platform for a wide array of information systems; outsourcing development
M	Lead-time	Time, cost	Shorten lead-time and decrease cost. Synchronize platform evolution with application projects
E	Extendibility	Expand, time	Shorten lead-time and increase product scope to increase market share. Include more types of sensors
N	Lead-time	Cost, time	Realize economies across product set

they can be divided into those who want to spend and those who want to save. Judging from the initiatives studied in this thesis, Companies K, L, M, E and N have primarily been focused on saving, either in development and maintenance cost, or in lead-time. Companies B, F, H and I have rather been spending, to get more or newer products onto the market. All companies have however struggled with doing both at the same time, primarily so Companies C, D, G and J. If we look at the companies from Chapter IX, their goals have been to focus on core competencies, and to save licensing and development costs. For those who have primarily been using open source software, it has also been a way to take control over future support and maintenance.

The prioritized quality attributes, such as the ones shown in Table 19, are said to drive the architectural choices (Bass *et al.*, 1998), or the architectural transformations (Bosch, 2000). The quality attributes of Table 19, taken from the case studies, are akin to the two quality attributes *modifiability* and *reusability* from Table 1 in Chapter I. A quality attribute classification with higher fidelity would be needed to get a more exact relation between the quality attributes of the case companies with architectural techniques such as those presented

by Bass *et al.* (1998). This thesis has focused on architectural efforts to enable reuse, which naturally is related to savings in development cost and lead-time. Benefits of reuse are modeled quantitatively in Chapter X. The initiatives the studied companies have implemented have however also combined reuse efforts with introduction of variability, which then has enabled also expansive business goals, to diversify product portfolios and enter new markets.

Question 2: How does the software architecture relate to the organization and the software process?

The clearest relationship between a software architecture and the organization that develops products from it can be seen in the way development projects are structured. Most organizations in this thesis divide their projects into teams according to the components of the architecture, with roles responsible for each module. This usually creates a matrix organization where resources dedicated to specific tasks such as requirements engineering, design, development and testing are taken from the line organization to populate projects. Individuals will then specialize in their tasks according to the various software process phases, but also specialize on the particular modules they are developing. Changes in the anatomy of the architecture will then lead to changes in the structure of teams within projects. However, alternative strategies of development exist, where focus is shifted over to the features of the product, or the functions to be performed in software development, rather than the modules of the product. The research in this thesis however seems to indicate that the stability of the architecture has impact on the choices that can be made in these issues. Chapter VI shows an initial framework for this linkage, while Chapter VII shows the impact of individual architectural changes, and Chapter VIII shows examples of when the whole strategy to development has to be changed.

Chapter VII also shows other linkages between software architectures, processes and organizations. Software architecture changes imply changes to the rules of how software is developed, and since the software process concerns how software is developed, the two have to be aligned when changes are made. A further challenge with this is that software developers are inclined to change *what* they develop, but not *how* they develop. Chapter VIII finally investigates how to organize for development of architectural assets which are used by several projects. Examples are shown of companies that organize development around these assets, and those that maintain their organizational focus on the products that are to be marketed. Some examples even show how the entire business of the company changes as internally developed assets are turned into products of their own.

Question 3: How do organizations carry out architectural changes, and how can this process be improved?

The process for architectural change is the focus of Chapter VII. It shows examples of architectural changes that are decided and performed as ordinary technical changes, but which face opposition as organizational issues are overlooked. A generic decision process is provided, which should help muster support for any change, and raise awareness of the impact on process and organiza-

tion such a change can have. Examples of different strategies for rolling out decided changes are also given, e.g. integrated with product-focused projects, or as standalone projects later merged into product code with resources from the line.

One challenge for software engineers when convincing management of the benefits of a suggested architectural investment is to present its business case. It can also be difficult to convince other developers to change their ways, and spend time and resources on what seems to be over-ambitious architectures. Chapter X presents one example of how a quantitative model can be designed and used to make the case for implementing a reuse-based architectural strategy.

Question 4: Which architectural strategies exist, with bearing on business goals related to productivity?

Chapter VIII gives examples of several architectural and development strategies, and shows a framework for how companies can achieve various business goals based on these strategies. Companies should not automatically strive for the highest possible level of architectural maturity, but rather asses their needs and goals with relation to opportunities for reuse and increased productivity. The dilemma of investing in architectural resources and reusable assets, with e.g. approaches such as software product lines, is that when implemented, they can provide shorter lead-times and quicker access to new markets – the implementation of such approaches can however delay release of products being currently developed. A company in such a situation therefore has to investigate the opportunities given by other approaches, such as the ones presented in Chapter VIII, or the light-weight approaches to software product lines, presented in Chapter IV and employed by one of the cases in Chapter VIII. Chapter IX further investigates how open source development can be used as a strategy to reach goals of decreasing development effort, focusing on core competencies, and gaining power in relation to partners and suppliers.

Question 5: How do software architectures evolve to continue supporting business goals in an evolving environment?

Bosch (2000) presents reasons for, and ways to evolve a software product line. These are on the levels of introducing a new product line, introducing new products into a product line, adding new features to the products of a product line, supporting new standards and infrastructures, and improving the quality attributes of the product line. The work presented here has instead been investigating when and how to change strategy to architecture and development altogether. These two views should however match.

Three generic strategies have been seen in the studies of this thesis: packaging technology – either found in products developed by a company, or gained by experience as a service company – in products for the market; maintaining a stable architecture for streamlined development; and packaging and maintaining reusable assets as internal components which can be used to develop a variety of products. A discussion of these three generic strategies follow.

This thesis shows examples of organizations that have gained a technological advantage, and packaged this technology in internal reusable assets. Such organizations may see this technology as their main asset, and divest proper prod-

uct development to instead market their technology as a product, or as a service. The benefit of a product is that it has extraordinary potential for revenues, compared to a service, where resources have to be spent on integrating licensed technology with customers' products. Such services will often bring revenues based on the effort to do this integration, rather than based on the value of the technology (Cusumano, 2004). The companies that in this thesis have managed to capitalize on this opportunity of packaging internal technology assets as products have done this from an architectural approach. The opposite has also been true, for organizations that have divested technology, and focused on core competencies closer to the end customer, such as added-value features, marketing and branding. These initiatives are easier to implement when there exist structured or open interfaces between internal technology and external features.

An alternative has for some companies been to keep reusable assets internal. With the right balance between these internal assets, and the products based on them, they have managed to develop products customized for individual customers, or differentiate their products for greater market coverage. This requires a proper organization, an adequate process to channel requirements from customers, through individual products, down to the internal assets, and strategies to prioritize resources between individual products and internal assets. This is a balance that is difficult to maintain, but that can provide great opportunities for customization and differentiation. The cost to maintain this balance might however be too high when market conditions change, and the organization must be open to changes in strategies for development.

Other organizations have had one main product, which has been released in continuous versions. The continuing ability to release such versions depends on the stability of the software architecture – it must allow for the changes of the product that the market demands. New versions must include enough new functionality to warrant upgrades from old customers, or to attract new ones. The architecture must also be maintained and adapted to continue facilitating new requirements. Here is also a balance that is difficult to maintain – taking resources for developing new features and place on maintenance of the architecture. This balance will eventually tip over, so companies in such a situation must be prepared to discontinue developing products from the old architecture, and make significant reengineering of their product, to allow for new market requirements.

1.2 Research Goal

The goal of this thesis was stated in Chapter II:

> Help developers and architects utilize and evolve their software architectures to better support the business goals of their organizations, with respect to quality attributes related to increased productivity.

The combined answers to the research questions provide us with guidance in fulfilling this goal. The thesis shows that opportunities exist for using architectural strategies to reach business goals to save resources and lead-time, but also to diversify product portfolios and thereby entering new market niches. Gaining support for such strategies can however be difficult, especially for cost saving strategies that require initial investments, or any strategy that risks delaying revenues. The thesis gives guidelines and techniques to alleviate this, by quantifying the benefits of architectural strategies such as flexibility and reusability, and by pointing out the necessity to investigate the total impact of such changes – not only to the products themselves, but also to the organizations developing them.

The thesis furthermore gives long-term strategic guidance on which strategies are beneficial for which evolutionary phases of a company or market. Initial markets imply focusing on individual customers. If a market catches on, focus should go back to further productifying the offering, to catch as much as possible of the market with one product that sets the infrastructure and standard for such a market. Later, more mature markets, require companies to yet again differentiate their products, with product varieties focused on certain market segments. In such varying phases of a market, architectural strategies must change. Focus on individual customers often require individual projects for each product, but similarities between such products can be extracted into internal reusable assets. Such generic assets can form the basis for a mass market product. If opportunities arise to differentiate this product, management of variability in the product can lead to a new phase of customization. Architectural efforts should not only adapt to such a changing environment, but can also lead this change. If this is to be successful, developers and architects must understand market opportunities and business goals, or have effective communication with marketing and business management within the organization.

2. Further Work

Further work for the individual studies has been presented in the respective chapters. Further work aiming at the overall goal of this thesis should be cross-disciplinary, with proper analysis of the business strategy of software developing companies, in parallel with analysis of the architectural strategies used by these companies. The work of this thesis, as it is now, has primarily been a bottom-up approach to utilizing the full opportunities of architectural strategies to reach business objectives related to productivity. It gives guidance on how architects should act to make an impact, and what options they have to support their organizations in varying business conditions.

Further work should therefore be done in cooperation with research in business administration. Proper studies of the business environments of companies such as the ones studied here could anchor or mediate the views held in this thesis with business management in such organizations. Architects struggling to

employ architectural strategies such as the ones presented here could then receive top-down support, to leverage the impact of architectural strategies.

Such studies could be combined with in-depth studies of practitioners who are subjected to these architectural changes, and have to develop software according to them. These studies could perhaps be performed as observational studies. They would allow us to minimize resistance to architectural changes from within an organization. The two viewpoints of business and developers would also greatly strengthen the credibility and validity of the results presented in this thesis, as a continuing search of evidence which disconforms to the theories that have emerged in this thesis (Glaser, 1992). The thesis has mainly focused on the perspective of an architect or similar role.

A common topic of the further work discussions in the various studies has been to actually implement suggestions. This type of action research (Patton, 2001) would require generation of more in-depth and tangible conclusions, to make suggestions about architectural change and evolution credible enough to base decisions on. It would also provide proper feedback to, and validation of the tools and processes for decision support presented in this thesis. The studies in Chapters VI and X generated concrete suggestion or decision support for the companies involved; in the concrete cases about which risks need to be mitigated first, which bottlenecks existed in the requirements process, and which components should be generalized for reuse first. The decisions made based on this and other information has however not been followed up in this thesis. Other studies in this thesis have foremost generated conclusions general to all involved companies, and not focused on supporting the individual companies.

References

Aaker, D. A. (2001), *Strategic Market Management,* 6th Ed., John Wiley & Sons: Hoboken, NJ.

Abowd, G., Bass, L., Clements, P., Kazman, R., Northrop, L. and Zaremski, A. (1997), *Recommended Best Industrial Practice for Software Architecture Evaluation*, CMU/SEI-96-TR-025.

Albrecht, A. J. (1979), "Measuring Application Development Productivity", *Proceedings of the SHARE/GUIDE IBM Applications Development Symposium*, Monterey, CA.

America, P., Hammer, D., Ionita, M. T., Obbink, H. and Rommes, E. (2005), "Scenario-based Decision Making for Architectural Variability in Product Families", *Software Process Improvement and Practice*, 2005(10), pp 171-187.

Anastasopoulos, M. and Gacek, C. (2001), "Implementing Product Line Variabilities", *Proceedings of the 2001 Symposium on Software Reusability*, Toronto, Ontario, May 2001.

Ares, J., García, R., Juristo, N., López, M. and Moreno, A. M. (2000), "A More Rigorous and Comprehensive Approach to Software Process Assessment", *Software Process: Improvement and Practice*, Vol. 5, No. 1, pp 3-30, John Wiley & Sons, Ltd.

Asundi, J. and Kazman, R. (2001), "A Foundation for the Economic Analysis of Software Architectures", *Proceedings of the 3rd International Workshop on Economics-Driven Software Engineering Research (EDSER-3)*, Toronto, Canada, May 2001.

Asundi, J., Kazman, R. and Klein, M. (2000), "An Architectural Approach to Software Cost Modeling", *Proceedings of the 2nd International Workshop on Economics-Driven Software Engineering Research*, Limerick, Ireland.

Bahsoon, R. and Emmerich, W. (2003), "ArchOptions: A Real Options Based Model for Predicting the Stability of Software Architectures", *Proceedings of the 5th Workshop on Economics-Driven Software Research (EDSER-5)*, Portland, Oregon, May 2003.

Bandinelli, S. C., Fuggetta, A., Lavazza, L., Loi, M. and Picco, G. P. (1995), "Modeling and Improving an Industrial Software Process", *IEEE Transactions on Software Engineering*, Vol. 21, No. 5, pp 440-455.

Banks, J., Carson, J. S. and Nelson, B. L. (1996), *Discrete-Event System Simulation*, 2nd Ed., Prentice Hall.

Basili, V. R. and Turner, A. J. (1975), "Iterative Enhancement: A Practical Technique for Software Development", *IEEE Transactions on Software Engineering*, Vol. 1, No 4.

Basili, V. R. and Rombach, H. D. (1988), "The TAME project: towards improvement-oriented software environments", *IEEE Transactions on Software Engineering*, Vol. 14, No. 6, pp 758-773.

Basili, V. R., Caldiera, G. and Rombach, H. D. (1994), "The Experience Factory", in Marciniak, J. J. (Ed.), *Encyclopedia of Software Engineering*, Vol. 1, pp 469-476, John Wiley & Sons, Inc., New York.

Bass, L., Clements, P. and Kazman, R. (1998), *Software Architecture in Practice*, Addison Wesley Longman, Inc., Reading MA.

Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T. and DeBaud, J.-M. (1999), "PuLSE™: A Methodology to Develop Software Product Lines", *5th ACM SIGSOFT Symposium on Software Reusability (SSR'99)*, Los Angeles, CA, May 1999.

Beck, K. (1999a), *Extreme Programming Explained: Embrace Change*, 1st Ed., Addison-Wesley.

Beck, K. (1999b), "Embracing Change with Extreme Programming", *IEEE Computer*, Vol. 32, No. 10, pp 70-77.

Beckman, D. and Rigby, J. (2003), *Foundations of Marketing*, 8th Ed., Nelson.

Bergman, B. and Klefsjö, B. (2004), *Quality from Customer Needs to Customer Satisfaction*, Studentlitteratur.

Bergquist, M. and Ljungberg, J. (2001), "The power of gifts: organizing social relationships in open source communities", *Information Systems Journal*, Vol. 11, No. 4, pp. 305-320.

BigLever (2005), www.biglever.com, visited October 2005.

Birk A., Heller G., John I., Schmid K., von der Massen T. and Müller K. (2003), Product Line Engineering: The State of the Practice. *IEEE Software,* Vol. 20, No. 6, pp 52-60.

Birrer, G. E. and Carrica, J. L. (1990), *Present Value Applications for Accountants and Financial Planners*, Quorum Books.

Black, F. and Scholes, M. (1973), "The Pricing of Options and Corporate Liabilities", *Journal of Political Economy*.

Boehm, B. W. (1981), *Software Engineering Economics*, Prentice Hall PTR.

Bosch, J. (2000), *Design & Use of Software Architectures: Adopting and Evolving a Product-Line Approach*, Pearson Education Limited, London.

Bosch J. (2002), "Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organizations", *Proceedings of the 2nd International Conference on*

Software Product Lines (SPLC2), Lecture Notes in Computer Science, vol. 2379, Chastek G J (Ed.). Springer: Heidelberg, Germany, pp 257-272.

Bosch, J. (2005), "Staged Adoption of Software Product Families", *Journal of Software Process Improvement and Practice*, No. 10, 2005, John Wiley & Sons, pp 125-142.

Caine, A. and Banks Pidduck, A. (2004), "f² COCOMO: Estimating Software Project Effort and Cost", *Proceedings of the 6th International Workshop on Economics-Driven Software Engineering Research*, Edinburgh, Scotland, May 2004.

Christensen, C. M. (1997), *The Innovator's Dilemma – When New Technologies Cause Great Firms to Fail*, Harvard Business School Press.

Clements, P. and Northorp, L. (2001), *Software Product Lines: Practices and Patterns*, Addison Wesley, Boston.

Clements, P. (2002), "Being Proactive Pays Off", *IEEE Software*, July/August 2002, pp 28-31.

Clements, P., Kazman, R. and Klein, M. (2002), *Evaluating Software Architectures*, Addison-Wesley.

Cooper, R. G. (1990), "Stage-Gate Systems: A New Tool for Managing New Products", *Business Horizons*, May-June 1990, pp 44-55.

Cooper, R. G. and Kleinschmidt, E. J. (1993), "Stage gate systems for new product success", *Marketing Management*, Vol. 1 No. 4 pp 20-30, American Marketing Association.

Cooper, R. G. (2000), "Winning with new Products: Doing it Right", *Ivey Business Journal*, Vol. 64, No. 6, pp 54-61.

Cusumano, M. A. (2004), *The Business of Software: What Every Manager, Programmer, and Entrepreneur Must Know to Thrive and Survive in Good Times and Bad*, Free Press.

D'Cruz, C. and Ports, K. (2003), "Strategic Analysis Tools for High Tech Marketing", *Proceedings of the Portland International Conference on Management of Engineering and Technology*, Portland, Oregon, pp 408-415.

Dijkstra, E W. (1972), "The Humble Programmer", *Communications of the ACM,* Vol. 15, No. 10, pp 895-866.

Dion, R. (1993), "Process Improvement and the Corporate Balance Sheet", *IEEE Software*, Vol. 10, No. 4, pp 28-35.

Dobrica, L. and Niemelä, E. (2002), "A Survey on Software Architecture Analysis Methods", *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp 638-653.

Evans, R., Park, S. and Alberts, H. (1997), "Decisions not Requirements: Decision-Centered Engineering of Computer-Based Systems", *Engineering of Computer-Based Systems (ECBS)*, Monterey, pp. 435-442.

Farbey, B., Land, F. F. and Targett, D. (1993), *How to Evaluate Your I/T Investment: A study of methods and practice*, Butterworth Heinemann, Oxford.

Faulk S. R., Harmon R. R. and Raffo D. M. (2000), "Value-Based Software Engineering (VBSE): A Value-Driven Approach to Product-Line Engineering", *Software Product Lines: Experience and Research Directions (The Kluwer International Series in Engineering and Computer Science*, Vol. 576), Donohoe P. (Ed.), Kluwer, Boston.

Feller, J. and Fitzgerald, B. (2003), *Understanding Open Source Software Development*, Adison-Wesley.

Fenton, N. E. and Pfleeger, S. L. (1998), *Software Metrics: A Rigorous and Practical Approach*, 2nd Ed., Course Technology.

Ferrin B. and Plank R. (2002), "Total cost of ownership models: An exploratory study", *Journal of Supply chain management*, Vol. 38, No. 3, pp 18-29.

Ford, D. N. and Sobek, D. K. (2005), "Adapting Real Options to New Product Development by Modeling the Second Toyota Paradox", *IEEE Transactions on Engineering Management*, Vol. 52, No. 2, pp 175-185.

Fullan, M. (1991), *The New Meaning of Educational Change*, 2nd Ed., Cassell, London.

Fusaro, P., El Emam, K. and Smith, B. (1997), "Evaluating the Interrater Agreement of Process Capability Ratings", *Proceedings of the 4th International Software Metrics Symposium*, Bieman J (Chair), IEEE, Washington, pp 2-11.

Gamma, E., Helms, R., Johnson, R. and Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA.

Garlan, D. and Shaw, M. (1996), *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall.

Glaser, B. G. (1992), *Basics of grounded theory analysis: emergence vs forcing*, Sociology Press, Mill Valley, CA.

Grant, R. M. (1998), *Contemporary Strategy Analysis*, Blackwell Publishers Inc.

Golden, B. (2004), *Succeeding with Open Source*, Addison-Wesley.

Halling, M., Biffl, S. and Grünbacher, P. (2004), "The Role of Valuation in Value-Based Software Engineering", *Proceedings of the 6th International Workshop on Economics-Driven Software Engineering Research (EDSER-6)*, Edinburgh, May 2004.

von Hippel, E (2005), *Democratizing Innovation*, MIT Press.

Hofmeister, C., Nord, R. and Soni, D. (1999), *Applied Software Architecture,* Addison-Wesley.

Hohensohn, H., Hang, J. (2003), "Product - and Service Related Business Models for Open Source Software", *Tagungsband Net.ObjectDays*, Erfurt, Germany, September 2003.

Hohmann, L. (2003), *Beyond Software Architecture: Creating and Sustaining Winning Solutions*, Addison-Wesley: Harlow, UK.

House, E. R. (1980), *Evaluating with validity*, Beverly Hills, Sage Publications.

Höst, M., Regnell, B., Natt och Dag, J., Nedstam, J., and Nyberg, C. (2001), "Exploring bottlenecks in market-driven requirements management processes with discrete event simulation", *Journal of Systems and Software,* Vol. 59, Elsevier Science Inc., pp 323-332.

Ince, D. C. and Hekmatpour, S. (1988), "An Approach to Automated Software Design Based on Product Metrics", *Software Engineering Journal*, Vol. 3, No. 2, pp. 53-56.

ISO 9001:1994, *Quality Systems: Model for quality assurance in design, development, production, installation and servicing,* International Organization for Standardization.

ISO/IEC 9126-1:2001, *Software Engineering – Product Quality – Part 1: Quality Model,* International Organization for Standardization.

ITU (1992), *Specification and Description Language (SDL)*, ITU-T Standard Z.100, International Telecommunication Union.

Jamieson, D., Vinsen, K. and Callender, G. (2004), "Measuring Software Costs: A New Perspective on a Recurring Problem", *Proceedings of the 6th International Workshop on Economics-Driven Software Engineering Research*, Edinburgh, Scotland, May 2004.

Jarke, M. and Kurki-Suonio, R. (1998), "Guest Editorial: Introduction to the Special Issue: Scenario Management", *IEEE Transaction on Software Engineering*, Vol. 24, No. 12, pp 1033-1035, IEEE, Washington.

Karlsson, E.-A. (Ed., 1995), *Software Reuse: A Holistic Approach*, John Wiley & Sons.

Karlström, D. and Runeson, P. (2005), "Combining Agile Methods with Stage-Gate Project Management", *IEEE Software*, Vol. 22 No. 3, pp43-49, IEEE.

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*, pp 81-90.

Kazman, R., Klein, M., Barbacci, M., Longstaff, T., Lipson, H. and Carriere, J. (1998), "The Architecture Tradeoff Analysis Method", *Proceedings of the 4th IEEE International Conference on Engineering of Complex Computer Systems*, pp 68-78.

King, P.J.B. (1990), *Computer and Communication Systems Performance Modelling*, Prentice Hall.

Kolakowski, L. (1972), *Positivist philosophy from Hume to the Vienna Circle*, Pelican Books, 1972.

Kotter, J. P. (1996), *Leading Change*, Harvard Business School Press.

Krishnamurthy S. (2002), "Cave or Community?: An Empirical Examination of 100 Mature Open Source Projects", *First Monday*, Vol. 6, No. 7.

Kruchten, P. B. (1995), "The 4+1 View Model of Architecture", *IEEE Software*, Vol. 12, No. 6, pp 42-50.

Krueger, C. W. (1992), "Software Reuse", *ACM Computing Surveys*, pp 131-183, June 1992.

Krueger, C W. (2001), "Easing the Transition to Software Mass Customization", *Proceedings of the 4th International Workshop on Product Family Engineering, Lecture Notes in Computer Science*, vol. 2290, van der Linden F (Ed.), Springer, Berlin, pp 282-293.

Krueger, C. W. (2002a), "Filling the Technology Void for Industrial Software Product Lines", *Proceedings of the 7th International Conference on Software Reuse*, Austin, TX, April 2002.

Krueger, C. W. (2002b), "Eliminating the Adoption Barrier", *IEEE Software*, pp 28-31, July/August 2002

Krueger, C W. (2003), "Towards a Taxonomy for Software Product Lines", *Proceedings of the 5th International Workshop on Product Family Engineering (Lecture Notes in Computer Science*, vol. 3014*)*, van der Linden, F. (Ed.), Springer, Berlin, pp 323-331.

Kuvaja, P., Similä, J., Krzanik, L., Bicego, A., Koch, G. and Saukkonen, S. (1994), *Software Process Assessment and Improvement: The BOOTSTRAP Approach*, Blackwell Publishers, Oxford.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E. and Turski, W. M. (1997), "Metrics and Laws of Software Evolution - The Nineties View", *Proceedings of the 4th International Software Metrics Symposium*, IEEE, pp 20-32.

Lehman, M. M. and Ramil, J. F. (2003), "Software Evolution – Background, Theory, Practice", *Information Processing Letters*, Vol. 88, pp 33-44, Elsevier.

Lehtola, L., Kauppinen, M. (2004) "Evaluation of Two Requirements Prioritization Methods in Product Development Projects", *Proceedings of the European Software Process Improvement Conference (EuroSPI 2004)*, Trondheim, Norway.

Levitt, T. M. (1986), *The Marketing Imagination*, Free Press.

Lindsay, M., Dennis, M. (2001), "Product Cannibalization and the Role of Prices", *Applied Economics*, Vol. 33, No. 14, pp 1785-1793, Routledge.

McCabe, T. J. (1982), "Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric," *National Bureau of Standards Special Publication No. 500-99*, December 1982.

McFeely, R. (1996), *IDEAL: A User's Guide for Software Process Improvement*, SEI/CMU.

McKenna, R. (1991), *Marketing is Everything*, Harvard Business Review.

Merriam-Webster (1998), *Merriam-Webster's Collegiate Dictionary*, 10th Ed., Merriam-Webster.

Mills, H. D., O'Neill, D., Linger, R. C, Dyer, M. and Quinnan, R. E. (1980), "The Management of Software Engineering", *IBM Systems Journal*, Vol. 19, No. 4, pp 414-477

Minichiello, V., Aroni, R., Timewell, E. and Alexander, L. (1990), *In-depth interviewing: researching people*, Melbourne, Victoria, Longman Cheshire.

Moore, G. E. (1965), "Cramming More Components onto Integrated Circuits", *Electronics*, Vol. 38, No. 8.

Moore, G. A. (1991), *Crossing the Chasm: Marketing and Selling Technology Products to Mainstream Customers*, HarperCollins Publishers Inc., New York, NY.

Moore, G. A. (1995), *Inside the Tornado: Marketing Strategies from Silicon Valley's Cutting Edge*, HarperBusiness.

Moore, M., Kazman, R., Klein, M. and Asundi, J. (2003), "Quantifying the Value of Architecture Design Decision: Lessons from the Field", *Proceedings of the 25th International Conference on Software Engineering (ICSE 2003)*, Portland, Oregon, IEEE.

Naur, P. and Randell, B. (Eds., 1968), *Software Engineering: Report of a conference sponsored by the NATO Science Committee*, Oct 1968, NATO Scientific Affairs Division, Brussels.

Nedstam, J. and Karlsson, E.-A. (2004), "Experiences from Architectural Evolution", *Proceedings of the 5th Australian Workshop on System and Software Architectures (AWSA2004)*, April 2004. Swinburne University of Technology: Melbourne, Australia, pp 1-4.

Nedstam, J (2004), "Finalizing a PhD Thesis in Architectural Evolution", *Proceedings of the 6th International Workshop on Economics-Driven Software Engineering (EDSER-6)*, May 2004. IEE, Herts, UK, pp 71-74.

Nejmeh, B. A. and Thomson, I. (2002), "Business-Driven Product Planning Using Feature Vectors and Increments", *IEEE Software,* Vol. 19, No. 6, pp 34-42.

Nelson, K. M., Nelson, H. J. and Ghods, M. (1997), "Technology Flexibility: Conceptualization, Validation, and Measurement", *Proceedings of the 30th Hawaii International Conference on System Sciences*, Vol. 3, pp 76-87, IEEE, Los Alamitos, California.

Noori, H. (1990), *Managing the Dynamics of New Technology*, Prentice Hall.

Noppen, J. and Aksit, M. (2004), "A Case Study on Optimization of Resource Distribution to cope with Unanticipated Changes in Requirements", *Proceedings of the 6th International Workshop on Economics-Driven Software Engineering Research*, Edinburgh, Scotland, May 2004.

Olofsson, J. (2003), "Upphovsrättsliga aspekter på licenser för fri programvara och öppen källkod – en analys av tillämpligheten i svensk rätt", The Swedish Law and Informatics Research Institute, Stockholm University, IRI report 2003:1.

Opdyke, W. F. and Johnson, R. E. (1993), "Creating Abstract Superclasses by Refactoring", *Proceedings of the 1993 ACM Conference on Computer Science*, Indianapolis, Indiana, pp 66-73.

OSI (2005), *The Open Source Definition*, http://www.opensource.org/docs/definition.php, visited October 2005.

Padberg, F. and Müller, M. (2004), "On the Impact of Warmup Phases on the Economics of Pair Programming", *Proceedings of the 6th International Workshop on Economics-Driven Software Engineering Research*, Edinburgh, Scotland.

Patton, M. Q. (2001), *Qualitative Research and Evaluation Methods*, 3rd Ed., Sage Publications.

Paulk, M. C., Curtis, B., Chrissis, M. B. and Weber, C. V. (1993), "Capability Maturity Model, Version 1.1", *IEEE Software*, Vol. 10, No. 4, pp 18-27.

Perry, D. E. (1994), "Issues in Process Architecture", *Proceedings of the 9th International Software Process Workshop*, pp 138-140.

Pfleeger, S. L. (1998), *Software Engineering: Theory and Practice*, Prentice Hall.

Pitkethly, R. (1997), "The Valuation of Patents", *Judge Institute Working Paper*, 21/97, The Judge Institute of Management Studies, Cambridge, England.

Poladian, V., Butler, S., Shaw, M. and Garlan, D. (2003), "Time is Not Money: The case for multi-dimensional accounting in value-based software engineering", *Proceedings of the 5th Workshop on Economics-Driven Software Engineering Research (EDSER-5)*, Portland, Oregon, May 2003.

Pride, W., M. and Ferrell, O., C. (2003), *Marketing: Concepts and Strategies*, 12th Ed., Houghton Mifflin Company.

Rajlich, V. T. and Bennett, K. H. (2000), "A Staged Model for the Software Life Cycle", *IEEE Computer*, Vol. 33, No. 7, pp. 66-71.

Raymond, E. S. (1998), *"*The Cathedral and the Bazaar*", First Monday*, Vol. 3, No. 3.

Regnell, B., Beremark, P. and Eklundh, O. (1998), "A Market-Driven Requirements Engineering Process - Results from an Industrial Process Improvement Programme", *Journal of Requirements Engineering*, Vol. 3, No. 2, pp 121-129, Springer-Verlag.

Regnell, B., Höst, M., Natt och Dag, J., Berenmark, P. and Hjelm, T. (2001), "An Industrial Case Study on Distributed Prioritisation in Market-Driven Requirements Engineering for Packaged Software", *Requirements Engineering Journal*, Vol. 6, No. 1, pp 51-62, Springer-Verlag Ltd., London.

Reifer, D. J. (2004), "Use of Real Options Theory to Value Software Trade Secrets", *Proceedings of the 6th International Workshop on Economics-Driven Software Engineering Research*, Edinburgh, Scotland, May 2004.

Robson, C. (2002), *Real World Research*, 2nd Ed., Blackwell Publishers Inc.

Rogers, E. M. (2003), *Diffusion of Innovations*, 5th Ed., Free Press.

Royce, W. W. (1970), "Managing the development of large software systems: concepts and techniques", *Proceedings of IEEE WESTCON*, Los Angeles, CA.

Sanders, J. and Curran, S. (1994), *Software Quality*, Addison-Wesley.

Saaty, T. L. (1980), *The Analytical Hierarchy Process*, McGraw-Hill.

Scriven, M. (1991), *Evaluation Thesaurus*, 4th Edition, Sage Publications, Thousand Oaks.

SEI (2005a), http://www.sei.cmu.edu/productlines/framework.html, visited October 2005.

SEI (2005b), http://www.sei.cmu.edu/productlines/sei_events.html, visited October 2005.

Seifert, T. and Wieland, T. (2003), "Prerequisites For Enterprises To Get Involved In Open Source Software Development", *Tagungsband Net.ObjectDays*, Erfurt, Germany, September 2003.

Sommerville, I., *Software Engineering*, 6th Ed., Addison-Wesley, Harlow, UK 2001

Soni, D., Nord, R., Hofmeister, C. (1995), "Software Architecture in Industrial Applications", *Proceedings of ICSE 1995*, Seattle, pp. 196-210

SourceForge (2005), http://sourceforge.net/index.php, visited October 2005.

SPIN-Syd (2005), http://www.spin-syd.org, visited October 2005.

Staples, M. and Hill, D. (2004), "Experiences Adopting Software Product Line Development without a Product Line Architecture", *Proceedings of the Asia-Pacific Software Engineering Conference 2004*, Busan, Korea, IEEE Computer Society Press.

Stern, C. W. and Stalk, G. (1998; Eds.), *Perspectives on Strategy from The Boston Consulting Group*, Wiley.

Stroustrup, B. (1997), *The C++ Programming Language*, 3rd Ed., Addison-Wesley, Reading, MA.

Suarez, P. and Patt, A. G. (2004), "Cognition, caution, and credibility: the risks of climate forecast application", *Risk, Decision and Policy*, Vol. 9, No. 1, pp 75-89, Routledge.

TickIT (1995), *The TickIT Guide: A Guide to Software Quality Management System Construction and Certification to ISO 9001*, British Standards Institution, 1995.

Vixie, P. (1999), "Software Engineering", *Open Sources: Voices from the Open Source Revolution*, O'Reilly & Associates, pp. 91-100.

Vähäniitty, J. (2004), "Product Portfolio Management in Small Software Product Businesses – a Tentative Research Agenda", *Proceedings of the 6th International Workshop on Economics-Driven Software Engineering Research*, Edinburgh, Scotland, May 2004.

Ward, A., Liker, J. K., Cristiano, J. J. and Sobek, D. K. (1995), "The Second Toyota Paradox: How delaying decisions can make better cars faster", *Sloan Management Review*, Vol. 36, No. 3, pp 43-61.

Wohlin, C. and Regnell, B. (1999), "Strategies for Industrial Relevance in Software Engineering Education", *Journal of Systems and Software*, Vol. 49, No. 2-3, pp 125-134. Elsevier Science BV, Amsterdam.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C, Regnell, B. and Wesslén, A. (2000), *Experimentation in Software Engineering: An Introduction*, Kluwer Academic Publishers.

Yin, B. H. and Winchester, J. W. (1978), "The Establishment and Use of Measures to Evaluate the Quality of System Designs", *Proceedings of the Software Quality and Assurance Workshop*, pp 45-52.

Zahran, S. (1998), *Software Process Improvement: Practical Guidelines for Business Success*, Addison-Wesley, Harlow, UK.

Reports on Communication Systems

101. **On Overload Control of SPC-systems**
Ulf Körner, Bengt Wallström, and Christian Nyberg, 1989.
CODEN: LUTEDX/TETS- -7133- -SE+80P

102. **Two Short Papers on Overload Control of Switching Nodes**
Christian Nyberg, Ulf Körner, and Bengt Wallström, 1990.
ISRN LUTEDX/TETS- -1010- -SE+32P

103. **Priorities in Circuit Switched Networks**
Åke Arvidsson, *Ph.D. thesis*, 1990.
ISRN LUTEDX/TETS- -1011- -SE+282P

104. **Estimations of Software Fault Content for Telecommunication Systems**
Bo Lennselius, *Lic. thesis*, 1990.
ISRN LUTEDX/TETS- -1012- -SE+76P

105. **Reusability of Software in Telecommunication Systems**
Anders Sixtensson, *Lic. thesis*, 1990.
ISRN LUTEDX/TETS- -1013- -SE+90P

106. **Software Reliability and Performance Modelling for Telecommunication Systems**
Claes Wohlin, *Ph.D. thesis*, 1991.
ISRN LUTEDX/TETS- -1014- -SE+288P

107. **Service Protection and Overflow in Circuit Switched Networks**
Lars Reneby, *Ph.D. thesis*, 1991.
ISRN LUTEDX/TETS- -1015- -SE+200P

108. **Queueing Models of the Window Flow Control Mechanism**
Lars Falk, *Lic. thesis*, 1991.
ISRN LUTEDX/TETS- -1016- -SE+78P

109. **On Efficiency and Optimality in Overload Control of SPC Systems**
Tobias Rydén, *Lic. thesis,* 1991.
ISRN LUTEDX/TETS- -1017- -SE+48P

110. **Enhancements of Communication Resources**
Johan M. Karlsson, *Ph.D. thesis*, 1992.
ISRN LUTEDX/TETS- -1018- -SE+132P

111. **On Overload Control in Telecommunication Systems**
Christian Nyberg, *Ph.D. thesis*, 1992.
ISRN LUTEDX/TETS- -1019- -SE+140P

112. **Black Box Specification Language for Software Systems**
Henrik Cosmo, *Lic. thesis*, 1994.
ISRN LUTEDX/TETS- -1020- -SE+104P

113. **Queueing Models of Window Flow Control and DQDB Analysis**
Lars Falk, *Ph.D. thesis*, 1995.
ISRN LUTEDX/TETS- -1021- -SE+145P

114. **End to End Transport Protocols over ATM**
Thomas Holmström, *Lic. thesis*, 1995.
ISRN LUTEDX/TETS- -1022- -SE+76P

115. **An Efficient Analysis of Service Interactions in Telecommunications**
Kristoffer Kimbler, *Lic. thesis*, 1995.
ISRN LUTEDX/TETS- -1023- -SE+90P

116. **Usage Specifications for Certification of Software Reliability**
Per Runeson, *Lic. thesis,* May 1996.
ISRN LUTEDX/TETS- -1024- -SE+136P

117. **Achieving an Early Software Reliability Estimate**
Anders Wesslén, *Lic. thesis*, May 1996.
ISRN LUTEDX/TETS- -1025- -SE+142P

118. **On Overload Control in Intelligent Networks**
Maria Kihl, *Lic. thesis,* June 1996.
ISRN LUTEDX/TETS- -1026- -SE+80P

119. **Overload Control in Distributed-Memory Systems**
Ulf Ahlfors, *Lic. thesis,* June 1996.
ISRN LUTEDX/TETS- -1027- -SE+120P

120. **Hierarchical Use Case Modelling for Requirements Engineering**
Björn Regnell, *Lic. thesis,* September 1996.
ISRN LUTEDX/TETS- -1028- -SE+178P

121. **Performance Analysis and Optimization via Simulation**
Anders Svensson, *Ph.D. thesis*, September 1996.
ISRN LUTEDX/TETS- -1029- -SE+96P

122. **On Network Oriented Overload Control in Intelligent Networks**
Lars Angelin, *Lic. thesis,* October 1996.
ISRN LUTEDX/TETS- -1030- -SE+130P

123. **Network Oriented Load Control in Intelligent Networks Based on Optimal Decisions**
Stefan Pettersson, *Lic. thesis,* October 1996.
ISRN LUTEDX/TETS- -1031- -SE+128P

124. **Impact Analysis in Software Process Improvement**
Martin Höst, *Lic. thesis,* December 1996.
ISRN LUTEDX/TETS- -1032- -SE+140P

125. **Towards Local Certifiability in Software Design**
Peter Molin, *Lic. thesis,* February 1997.
ISRN LUTEDX/TETS- -1033- -SE+132P

126. **Models for Estimation of Software Faults and Failures in Inspection and Test**
Per Runeson, *Ph.D. thesis*, January 1998.
ISRN LUTEDX/TETS- -1034- -SE+222P

127. **Reactive Congestion Control in ATM Networks**
Per Johansson, *Lic. thesis,* January 1998.
ISRN LUTEDX/TETS- -1035- -SE+138P

128. **Switch Performance and Mobility Aspects in ATM Networks**
Daniel Søbirk, *Lic. thesis,* June 1998.
ISRN LUTEDX/TETS- -1036- -SE+91P

129. **VPC Management in ATM Networks**
Sven-Olof Larsson, *Lic. thesis,* June 1998.
ISRN LUTEDX/TETS- -1037- -SE+65P

130. **On TCP/IP Traffic Modeling**
Pär Karlsson, *Lic. thesis,* February 1999.
ISRN LUTEDX/TETS- -1038- -SE+94P

131. **Overload Control Strategies for Distributed Communication Networks**
Maria Kihl, *Ph.D. thesis*, March 1999.
ISRN LUTEDX/TETS- -1039- -SE+158P

132. **Requirements Engineering with Use Cases – a Basis for Software Development**
Björn Regnell, *Ph.D. thesis*, April 1999.
ISRN LUTEDX/TETS- -1040- -SE+225P

133. **Utilisation of Historical Data for Controlling and Improving Software Development**
Magnus C. Ohlsson, *Lic. thesis,* May 1999.
ISRN LUTEDX/TETS- -1041- -SE+146P

134. **Early Evaluation of Software Process Change Proposals**
Martin Höst, *Ph.D. thesis*, June 1999.
ISRN LUTEDX/TETS- -1042- -SE+193P

135. **Improving Software Quality through Understanding and Early Estimations**
Anders Wesslén, *Ph.D. thesis*, June 1999.
ISRN LUTEDX/TETS- -1043- -SE+242P

136. **Performance Analysis of Bluetooth**
Niklas Johansson, *Lic. thesis,* March 2000.
ISRN LUTEDX/TETS- -1044- -SE+76P

137. **Controlling Software Quality through Inspections and Fault Content Estimations**
Thomas Thelin, *Lic. thesis,* May 2000
ISRN LUTEDX/TETS- -1045- -SE+146P

138. **On Fault Content Estimations Applied to Software Inspections and Testing**
Håkan Petersson, *Lic. thesis,* May 2000.
ISRN LUTEDX/TETS- -1046- -SE+144P

139. **Modeling and Evaluation of Internet Applications**
Ajit K. Jena, *Lic. thesis,* June 2000.
ISRN LUTEDX/TETS- -1047- -SE+121P

140. **Dynamic traffic Control in Multiservice Networks – Applications of Decision Models**
Ulf Ahlfors, *Ph.D. thesis*, October 2000.
ISRN LUTEDX/TETS- -1048- -SE+183P

141. **ATM Networks Performance – Charging and Wireless Protocols**
Torgny Holmberg, *Lic. thesis,* October 2000.
ISRN LUTEDX/TETS- -1049- -SE+104P

142. **Improving Product Quality through Effective Validation Methods**
Tomas Berling, *Lic. thesis,* December 2000.
ISRN LUTEDX/TETS- -1050- -SE+136P

143. **Controlling Fault-Prone Components for Software Evolution**
Magnus C. Ohlsson, *Ph.D. thesis*, June 2001.
ISRN LUTEDX/TETS- -1051- -SE+218P

144. **Performance of Distributed Information Systems**
Niklas Widell, *Lic. thesis,* February 2002.
ISRN LUTEDX/TETS- -1052- -SE+78P

145. **Quality Improvement in Software Platform Development**
Enrico Johansson, *Lic. thesis,* April 2002.
ISRN LUTEDX/TETS- -1053- -SE+112P

146. **Elicitation and Management of User Requirements in Market-Driven Software Development**
Johan Natt och Dag, *Lic. thesis,* June 2002.
ISRN LUTEDX/TETS- -1054- -SE+158P

147. **Supporting Software Inspections through Fault Content Estimation and Effectiveness Analysis**
Håkan Petersson, *Ph.D. thesis*, September 2002.
ISRN LUTEDX/TETS- -1055- -SE+237P

148. **Empirical Evaluations of Usage-Based Reading and Fault Content Estimation for Software Inspections**
Thomas Thelin, *Ph.D. thesis*, September 2002.
ISRN LUTEDX/TETS- -1056- -SE+210P

149. **Software Information Management in Requirements and Test Documentation**
Thomas Olsson, *Lic. thesis*, October 2002.
ISRN LUTEDX/TETS- -1057- -SE+122P

150. **Increasing Involvement and Acceptance in Software Process Improvement**
Daniel Karlström, *Lic. thesis*, November 2002.
ISRN LUTEDX/TETS- -1058- -SE+125P

151. **Changes to Processes and Architectures; Suggested, Implemented and Analyzed from a Project Viewpoint**
Josef Nedstam, *Lic. thesis*, November 2002.
ISRN LUTEDX/TETS- -1059- -SE+124P

152. **Resource Management in Cellular Networks -Handover Prioritization and Load Balancing Procedures**
Roland Zander, *Lic. thesis*, March 2003.
ISRN LUTEDX/TETS- -1060- -SE+120P

153. **On Optimisation of Fair and Robust Backbone Networks**
Pål Nilsson, *Lic. thesis*, October 2003.
ISRN LUTEDX/TETS- -1061- -SE+116P

154. **Exploring the Software Verification and Validation Process with Focus on Efficient Fault Detection**
Carina Andersson, *Lic. thesis*, November 2003.
ISRN LUTEDX/TETS- -1062- -SE+134P

155. **Improving Requirements Selection Quality in Market-Driven Software Development**
Lena Karlsson, *Lic. thesis*, November 2003.
ISRN LUTEDX/TETS- -1063- -SE+132P

156. **Fair Scheduling and Resource Allocation in Packet Based Radio Access Networks**
Torgny Holmberg, *Ph.D. thesis*, November 2003.
ISRN LUTEDX/TETS- -1064- -SE+187P

157. **Increasing Product Quality by Verification and Validation Improvements in an Industrial Setting**
Tomas Berling, *Ph.D. thesis*, December 2003.
ISRN LUTEDX/TETS- -1065- -SE+208P

158. **Some Topics in Web Performance Analysis**
Jianhua Cao, *Lic. thesis*, June 2004.
ISRN LUTEDX/TETS- -1066- -SE+99P

159. **Overload Control and Performance Evaluation in a Parlay/OSA Environment**
Jens K. Andersson, Lic. thesis, August 2004.
ISRN LUTEDX/TETS- -1067- -SE+100P

160. **Performance Modeling and Control of Web Servers**
Mikael Andersson, *Lic. thesis*, September 2004.
ISRN LUTEDX/TETS- -1068- -SE+105P

161. **Integrating Management and Engineering Processes in Software Product Development**
Daniel Karlström, *Ph.D. thesis*, December 2004.
ISRN LUTEDX/TETS- -1069- -SE+230P

162. **Managing Natural Language Requirements in Large-Scale Software Development**
Johan Natt och Dag, *Ph.D. thesis*, Febuary 2005.
ISRN LUTEDX/TETS- -1070- -SE+222P

163. **Designing Resilient and Fair Multi-layer Telecommunication Networks**
Eligijus Kubilinskas, *Lic. thesis*, Febuary 2005.
ISRN LUTEDX/TETS- -1071- -SE+136P

164. **Internet Access and Performance in Ad hoc Networks**
Anders Nilsson, *Lic. thesis*, April 2005.
ISRN LUTEDX/TETS- -1072- -SE+119P

165. **Active Resource Management in Middleware and Service-oriented Architectures**
Niklas Widell, *Ph.D. thesis*, May 2005.
ISRN LUTEDX/TETS- -1073- -SE+162P

166. **Quality Improvement with Focus on Performance in Software Platform Development**
Enrico Johansson, *Ph.D. thesis*, June 2005.
ISRN LUTEDX/TETS- -1074- -SE+P139

167. **On Inter-System Handover in a Wireless Hierarchical Structure**
Henrik Persson, *Lic. thesis*, September 2005.
ISRN LUTEDX/TETS- -1075- -SE+90P

168. **Prioritization Procedures for Resource Management in Cellular Networks**
Roland Zander, *Ph.D. thesis*, November 2005.
ISRN LUTEDX/TETS- -1076- -SE+181P

169. **Strategies for Management of Architectural Change and Evolution**
Josef Nedstam, *Ph.D. thesis*, December 2005.
ISRN LUTEDX/TETS- -1077- -SE+192P