Mälardalen University Press Licentiate Theses
No. 97

SOFTWARE ARCHITECTURE EVOLUTION AND SOFTWARE EVOLVABILITY

Hongyu Pei Breivold

2009

MÄLARDALEN UNIVERSITY
SWEDEN

School of Innovation, Design and Engineering

ISSN 1651-9256
ISBN 978-91-86135-15-7
Printed by Arkitektkopia, Västerås, Sweden

Abstract

Software is characterized by inevitable changes and increasing complexity, which in turn may lead to huge costs unless rigorously taking into account change accommodations. This is in particular true for long-lived systems. For such systems, there is a need to address evolvability explicitly during the entire lifecycle, carry out software evolution efficiently and reliably, and prolong the productive lifetime of the software systems.

In this thesis, we study evolution of software architecture and investigate ways to support this evolution. The central theme of the thesis is how to analyze software evolvability, i.e. a system's ability to easily accommodate changes. We focus on several particular aspects: (i) what software characteristics are necessary to constitute an evolvable software system; (ii) how to assess evolvability in a systematic manner; (iii) what impacts need to be considered given a certain change stimulus that results in potential requirements the software architecture needs to adapt to, e.g. ever-changing business requirements and advances of technology.

To improve the capability in being able to on forehand understand and analyze systematically the impact of a change stimulus, we introduce a software evolvability model, in which subcharacteristics of software evolvability and corresponding measuring attributes are identified. In addition, a further study of one particular measuring attribute, i.e. modularity, is performed through a dependency analysis case study.

We introduce a method for analyzing software evolvability at the architecture level. This is to ensure that the implications of the potential improvement strategies and evolution path of the software architecture are analyzed with respect to the evolvability subcharacteristics. This method is proposed and piloted in an industrial setting.

The fact that change stimuli come from both technical and business perspectives spawns two aspects that we also look into in this research, i.e. to respectively investigate the impacts of technology-type and business-type of change stimuli.

Acknowledgements

My heartfelt thanks go to my main supervisor Prof. Ivica Crnkovic for believing in me, and for making the creation of this thesis a thoroughly constructive and enjoyable experience. You are a great supervisor with a great sense of humour, and you have been unfailingly generous with your time and your knowledge, giving me good advice and support when it is needed.

Many thanks go also to my assistant supervisors Prof. Magnus Larsson and Dr. Rikard Land, my industrial mentor Dr. Stig Larsson, for your constant support and encouragement throughout this work. I also appreciate the opportunities given by Prof. Magnus Larsson and Dr. Fredrik Ekdahl, introducing me to the journey of research. Very special thanks to Prof. Judith Stafford, Prof. Nenad Medvidović and Prof. Michel Chaudron for advice and suggestions in the beginning of my research.

I am grateful to the best team of reviewers, who made time in their very busy schedules to read and comment on my drafts. I give my sincerest thanks to each of them, who deserve special recognition for their unique insights and commentary: Prof. Ivica Crnkovic, Dr. Rikard Land, Dr. Stig Larsson, Prof. Magnus Larsson, Dr. Anders Wall, Dr. Daniel Sundmark, Peter Eriksson, Dr. Fredrik Ekdahl and Chuck Connell. Their careful reading and practical suggestions have led to great improvements of this work.

I would also like to thank Prof. Hans Hansson for guidance in research planning, Dr. Gordana Dodig-Crnkovic and Dr. Jan Gustafsson for introducing me to the research methodology, Dr. Thomas Nolte for advice on networking and research in general, Harriet Ekwall and Monica Wasell for helping out. Many thanks go also to colleagues from ABB, people from the SAVE-IT industrial graduate school and BESS (Business oriented Engineering of Software intensive Systems) research group for nice company and discussions. Additionally, the work would not have been possible without the support from ABB Corporate Research and KKS, providing me with opportunities and resources for the research study.

I have been lucky to get to know a group of smart and energetic people who have given much joy and moral support. I especially want to thank Séverine Sentilles, Aneta Vulgarakis, Dr. Pasqualina Potena, Dr. Cristina Seceleanu, Dr. Tiberiu Seceleanu, Hüseyin Aysan, Moris Behnam, Yue Lu, Farhang Nemati, Marcelo Santos, Iva Krasteva, Dr. Mikael Åkerholm, Dr. Dag Nyström, Stefan Bygde, Anna Östholm, Yina Zhang and Chenyang Steen for your friendship and nice company.

This work would not be possible without the support of my family. I especially want to thank my parents for showing me the truths of love, gentleness, courage and persistence. Thanks to my brother for always caring about me and supporting me. I want also to express my immense appreciation to Anita Sletmo, Lasse Sletmo and Stig Lundvall, who have become one inseparable part of our family through years of deep and genuine friendship. Thank you so much for all the tremendous help and my gratitude to you cannot be summarized in a few words alone. Finally, I would like to dedicate this work to my beloved husband and my wonderful children, who have been a source of motivation and inspiration for me all along. Thanks Jon - for your love, patience, encouragement and continued support. Thanks Johanna, Martin and Elin - you are my sunshine!

Hongyu Pei Breivold

Linz, November, 2008

List of Included Papers

Paper A	*Analyzing Software Evolvability*, Hongyu Pei Breivold, Ivica Crnkovic, Peter J. Eriksson, Proceedings of the 32nd IEEE International Computer Software and Applications Conference (COMPSAC), Turku, Finland, July, 2008
Paper B	*Analyzing Software Evolvability of an Industrial Automation Control System: A Case Study*, Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land, Magnus Larsson, Proceedings of the 3rd International Conference on Software Engineering Advances (ICSEA), IEEE, Sliema, Malta, October, 2008
Paper C	*Using Dependency Model to Support Software Architecture Evolution*, Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land, Stig Larsson, Proceedings of the 4th International ERCIM Workshop on Software Evolution and Evolvability (Evol'08) at the 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering, IEEE, L'Aquila, Italy, September, 2008
Paper D	*Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles*, Hongyu Pei Breivold, Magnus Larsson, Proceedings of the 33rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Component Based Software Engineering (CBSE) Track, IEEE, Lübeck, Germany, 2007
Paper E	*Migrating Industrial Systems towards Software Product Lines: Experiences and Observations through Case Studies*, Hongyu Pei Breivold, Stig Larsson, Rikard Land, Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement (SPPI) Track, IEEE, Parma, Italy, September, 2008

Full List of Publications

Conferences and Workshops

- *Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles*, Hongyu Pei Breivold, Magnus Larsson, 33rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Component-Based Software Engineering (CBSE) Track, IEEE, Lübeck, Germany, August, 2007
- *Evaluating Software Evolvability*, Hongyu Pei Breivold, Ivica Crnkovic, Peter Eriksson, 7th Conference on Software Engineering and Practice in Sweden (SERPS), Göteborg, Sweden, October, 2007
- *Analyzing Software Evolvability*, Hongyu Pei Breivold, Ivica Crnkovic, Peter J. Eriksson, 32nd IEEE International Computer Software and Applications Conference (COMPSAC), Turku, Finland, July, 2008
- *Migrating Industrial Systems towards Software Product Lines: Experiences and Observations through Case Studies*, Hongyu Pei Breivold, Stig Larsson, Rikard Land, 34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement (SPPI) Track, IEEE, Parma, Italy, September, 2008
- *Using Dependency Model to Support Software Architecture Evolution*, Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land, Stig Larsson, 4th International ERCIM Workshop on Software Evolution and Evolvability (Evol'08) at the 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering, IEEE, L'Aquila, Italy, September, 2008
- *Analyzing Software Evolvability of an Industrial Automation Control System: A Case Study*, Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land, Magnus Larsson, 3rd International

Conference on Software Engineering Advances (ICSEA), IEEE, Sliema, Malta, October, 2008

Technical Report

- *Using Software Evolvability Model for Evolvability Analysis*, Hongyu Pei Breivold, Ivica Crnkovic, Technical Report ISSN 1404-3041 ISRN MDH-MRTC-222/2008-1-SE, Mälardalen Real-Time Research Center (MRTC), Mälardalen University, February, 2008

Tutorial

- *Emerging Technologies in Industrial Context – Component-Based and Service-Oriented Software Engineering*, Ivica Crnkovic, Hongyu Pei Breivold, 31st IEEE International Computer Software and Applications Conference (COMPSAC), Beijing, China, July, 2007

Table of Contents

Part 1

Chapter 1. Introduction

For long-lived industrial software, the largest part of lifecycle costs is concerned with the evolution of software to meet changing requirements [Bennett 1996]. There is a need to change software on a constant basis with major enhancements within a short timescale in order to keep up with new business opportunities. This puts critical demands on the software system's capability of rapid modification and enhancement to achieve cost-effective software evolution.

[Lehman et al. 2000] describes two views on software evolution: *what* and *why* versus the *how* perspectives. The former perspective studies the nature of the software evolution phenomenon and investigates its driving factors and impacts. The latter perspective studies the pragmatic aspects, i.e. technology, methods and tools that provide the means to control software evolution. In this research, we focus on the *how* perspective of software evolution.

According to [Madhavji et al. 2006], the term evolution reflects "*a process of progressive change in the attributes of the evolving entity or that of one or more of its constituent elements. What is accepted as progressive must be determined in each context. It is also appropriate to apply the term evolution when long-term change trends are beneficial, i.e. value or fitness is increasing over time, and more adapted to a changing environment even though isolated or short sequences of changes may appear degenerative.*" Specifically, software evolution relates to how software systems evolve over time [Yu et al. 2008]. It is one term that expresses the software changes during software system's lifecycle.

One of the principle challenges in software evolution is the ability to evolve software over time to meet the changing requirements of its stakeholders [Nehaniv and Wernick 2007]. In this context, software evolvability is an attribute that describes the software system's capability to accommodate changes. To better explain the term evolvability, we refer to the definition of *Software Evolvability* in [Rowe et al. 1994]:

"Software evolvability is an attribute that bears on the ability of a system to accommodate changes in its requirements throughout the system's lifespan with the least possible cost while maintaining architectural integrity"

1.1 Research Motivation

The evolution of software systems is characterized by inevitable changes and increasing complexity, which in turn may lead to huge costs unless rigorously taking into account change accommodations. This is in particular true for long-lived systems.

The focus of our research is primarily aimed at analyzing software evolvability for embedded industrial systems that often have a lifetime of 10-30 years. These systems are subject to and may undergo a substantial amount of evolutionary changes, e.g. software technology changes, system migration to product line architecture, ever-changing managerial issues such as demands for distributed development, and ever-changing business decisions driven by market situations. Therefore, for such long-lived systems, there is a need to address evolvability explicitly during the entire lifecycle, carry out software evolution efficiently and reliably, and prolong the productive lifetime of the software systems. As software architecture holds a key to the possibility to implement changes in an efficient manner [Bass et al. 2003], software architecture evolution becomes a critical part of the software lifecycle.

According to [Weiderman et al. 1997], software evolvability is a fundamental element for increasing strategic decisions, characteristics, and economic value of the software. Thus, the need for greater system evolvability is becoming recognized [Rowe and Leaney 1997]. We have also observed this need from various cases in industrial context [Breivold et al. 2008; Christian 2006], where evolvability was identified as a very important quality attribute that must be maintained. However, to our knowledge, there are no systematic means for evaluating the evolvability of a system and thus no means to analyze and compare software systems in terms of evolvability. Therefore, the motivation of this thesis is to build up a software evolvability model and to investigate ways to analyze the ability to evolve software.

In this thesis, we describe and make contributions to the following aspects:

1. Identify characteristics that are necessary for the evolvability of a software system;

2. Assess software evolvability in a systematic manner;
3. Investigate means for quantitatively assessing quality impact through using specific quality metrics;
4. Analyze the corresponding impacts, given a certain type of change stimulus.

1.2 Research Questions

We describe in the previous section that software architecture evolution is a critical part of software lifecycle, and that there is a need to explicitly address software evolvability. Therefore, the overall question of this thesis is:

How to analyze the evolvability of a software system?

Before we can determine how to analyze software evolvability, we need to understand what characteristics of software constitute the evolvability of a software system, i.e. what characteristics of software make it easier to change a software system as requirements evolve. To this end, we formulate the following research question which provides a starting point for further research:

What subcharacteristics are of primary importance for the evolvability of a software system? **(Q1)**

Once we know what subcharacteristics are of primary importance for the evolvability of a software system, we would like to have the means to assess software evolvability. Thus, the next question relates to the assessment of software evolvability in terms of subcharacteristics:

How can software evolvability be assessed in a systematic manner? **(Q2)**

According to [Yang and Ward 2003], software evolvability concerns both business and technical perspectives, as the stimuli of changes in software evolution can be related to both. Any change stimulus results in a collection of potential requirements that the software architecture needs to adapt to. Some examples of change stimuli are changes in environment, organization, process, technology and stakeholders' needs. These change stimuli have impact on the software system in terms of software architecture and its quality attributes. Thus, the next question relates to the impact analysis of a given change stimulus:

Given a certain type of change stimulus, what kind of impacts need to be considered? **(Q3)**

1.2.1 Detailed Studies

Detailed studies have been performed with respect to the research questions Q1 and Q3. We describe in this section the more detailed and specific research questions that are relevant to Q1 and Q3.

As a continuation of the first research question Q1, one additional contribution of the thesis is a deeper study of one of the measuring attributes identified in the answer to the first research question. Part of the answer to Q1 is an evolvability model which refines software evolvability into a collection of subcharacteristics that can be measured through a number of measuring attributes. The next research question is a continuation of Q1 and further explores one particular measuring attribute, i.e. modularity. The choice of focusing on software modularity is motivated mainly by the fact that modularity affects the behavior of a design with respect to most of the evolvability subcharacteristics, and that not much data has been published with respect to large scale industrial software systems [LaMantia et al. 2008]. This leads to the following detailed research question:

What modularization means can be used to support software architecture evolution? **(Q1.1)**

To answer the research question Q3, we have performed two case studies that represent two different types of change stimuli, i.e. technology-type and business-type. This is due to the fact that software evolvability concerns both technical and business issues [Yang and Ward 2003]. Thus we look into both technical and business aspects. These two aspects are further expressed through the subsequent two detailed research questions Q3.1 and Q3.2.

(1) Investigate the impact of technology-type change stimuli

With frequent advances in software engineering, the need to evolve software arises. As a consequence, software evolution faces different problems and challenges as new technologies are introduced. It has been witnessed that designing and implementing a large scale and complex system is a challenging task [Crnkovic and Larsson 2002]. In this thesis, we focus on two of the most well recognized software engineering paradigms coping with this challenge, i.e. component-based software engineering (CBSE) and service-oriented software engineering (SOSE). Thus, the next question relates to the impact analysis of the advances of technological paradigms:

> *Given the technology-type change stimulus of introducing SOSE to CBSE, what impacts need to be considered?* **(Q3.1)**

(2) Investigate the impact of business-type change stimuli

One of the main difficulties of software evolution is that all artifacts produced and used during the entire software lifecycle are subject to changes [Mens and Demeyer 2008]. Meanwhile, to keep up with new business opportunities, the need for differentiation in the marketplace, with short time-to-market as part of the need, has put critical demands on the effectiveness of software reuse. In this context, the change stimuli come from the business perspective. Accordingly, software product line approach has emerged as one specific type of software evolution, and has become one of the most established strategies for achieving large-scale software reuse and ensuring rapid development of new products [Birk et al. 2003]. However, product line development seldom starts from scratch. Instead, it is very often based on existing legacy implementations [Kotonya and Hutchinson 2008], and the issue of keeping legacy systems operational becomes critical. Accordingly, an important and challenging type of software evolution is how to cost-effectively manage the migration of legacy systems towards product lines. This leads to the following research question:

> *Given the business-type change stimulus of adopting a product line approach, what impacts need to be considered from a software evolution perspective?* **(Q3.2)**

1.3 Thesis Overview

The thesis is divided into two parts. The first part comprises a summary of the research. Chapter 1 describes the background, motivation and research questions of the performed research. Chapter 2 describes the research results, by recapitulating the research questions. Chapter 3 discusses the method used and the validity of the presented research. Chapter 4 surveys related work. Chapter 5 concludes the thesis and outlines future work that formulates potential tracks for further PhD studies.

The second part of this thesis is a collection of peer-reviewed conference and workshop papers that document details of the answers to the research questions, methods, and results. The following papers are included in this part:

Paper A "Analyzing Software Evolvability". Hongyu Pei Breivold, Ivica Crnkovic, Peter J. Eriksson. *Proceedings of the 32nd IEEE International Computer Software and Applications Conference (COMPSAC)*, Turku, Finland, July, 2008.

This paper contributes to the answer to the first research question Q1. The paper describes the initial establishment of an evolvability model as a framework for the analysis of software evolvability. We motivate and exemplify the model through an industrial case study of a software-intensive automation system.

I was the main author and contributed with the proposed evolvability model and the case study. The coauthors contributed with advices regarding the research method, discussions regarding the analysis and reviews.

Paper B "Analyzing Software Evolvability of an Industrial Automation Control System: A Case Study". Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land, Magnus Larsson. *Proceedings of the 3rd International Conference on Software Engineering Advances (ICSEA), IEEE*, Sliema, Malta, October, 2008.

This paper contributes to the answer to the second research question Q2. The paper describes our work in analyzing software evolvability of an industrial automation control system, and presents 1) evolvability subcharacteristics based on the problems in the case and available literature; 2) a structured method for analyzing software evolvability at the architectural level - the ARchitecture Evolvability Analysis (AREA) method. This paper includes also the main analysis results and our observations during the evolvability analysis process in the case study.

I was the main author and contributed with the description of the proposed evolvability analysis method, the case study, the analysis results and conclusions. The coauthors contributed with advice regarding research method, discussions regarding the analysis and reviews.

Paper C "Using Dependency Model to Support Software Architecture Evolution". Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land, Stig Larsson. *Proceedings of the 4th International ERCIM Workshop on Software Evolution and Evolvability (Evol'08) at the 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering, IEEE*, L'Aquila, Italy, September, 2008.

This paper contributes to the answer to the research question Q1.1. The paper explores the relationships between software evolvability, modularity and inter-module dependency, as designing software for ease of extension and contraction depends on how well the software structure is organized. Through a case study of an industrial power control and protection system, we describe our work in managing its software architecture evolution, guided by the static dependency analysis at the architectural level. The paper includes also the main analysis results, experiences and reflections during the dependency analysis process in the case study.

I was the main author and led the case study. I contributed with the description of managing software architecture evolution using the dependency analysis results as inputs, as well as the analysis and conclusions. The coauthors contributed with advice regarding the case description and reviews.

Paper D "Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles". Hongyu Pei Breivold, Magnus Larsson. *Proceedings of the 33rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Component Based Software Engineering (CBSE) Track, IEEE*, Lübeck, Germany, 2007.

This paper contributes to the answer to the research question Q3.1. The paper describes a comparison analysis framework of Component-Based Software Engineering (CBSE) and Service-Oriented Software Engineering (SOSE), and analyzes them from a variety of perspectives. We discuss as well the possibility of combining the strengths of the two engineering paradigms for improved quality attributes. This paper clarifies the characteristics of CBSE and SOSE, tries to shorten the gap between them and bring the two worlds together so that researchers and practitioners become aware of essential issues of both paradigms. Clarifying the characteristics of CBSE and SOSE may serve as inputs for further utilizing them in a reasonable and complementary way.

I was the main author and contributed with the comparison analysis framework, the analysis and conclusions. The coauthor contributed with advice and discussions regarding the analysis and reviews. In addition, Prof. Ivica Crnkovic contributed with valuable feedback and comments through reviews.

Paper E "Migrating Industrial Systems towards Software Product Lines: Experiences and Observations through Case Studies". Hongyu Pei Breivold, Stig Larsson, Rikard Land. *Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement (SPPI) Track, IEEE*, Parma, Italy, September, 2008.

This paper contributes to the answer to the research question Q3.2. The paper presents a product line migration method and describes our experiences in migrating industrial legacy systems into product lines. The migration method focuses on the migration process when the migration decision has been made. In addition, we present a number of recommendations for the transition process. They are of value to organizations that are considering a product line approach to their business. The recommendations cover four perspectives, i.e. business, organization, product development processes and technology.

I was the main author and contributed with the description of recommended practices in product line migration, the analysis and conclusions. The coauthors contributed with advice regarding research method and reviews.

In addition, the following report is indirectly related to the thesis. Part of the results from this report has been used in the preparation of part 1 of this thesis:

- "Using Software Evolvability Model for Evolvability Analysis", Hongyu Pei Breivold, Ivica Crnkovic, *Technical Report ISSN 1404-3041 ISRN MDH-MRTC-222/2008-1-SE, Mälardalen Real-Time Research Center, Mälardalen University*, February, 2008 [Breivold and Crnkovic 2008]

Chapter 2. Research Results

This chapter provides a brief overview the research results. The details are presented in the appended papers in the second part of the thesis.

We describe in section 1.2 that the overall question motivating the thesis is:

How to analyze the evolvability of a software system?

We further refine this question into several concrete research questions. For each of these questions, we present an answer here and relate the research questions with the individual papers included in this thesis.

What subcharacteristics are of primary importance for the evolvability of a software system? **(Q1)**

The subcharacteristics that are of primary importance for software evolvability in a given context (long-lived software-intensive systems) are described in paper A and B: ***Analyzability***, ***Architectural Integrity***, ***Changeability***, ***Extensibility***, ***Portability***, ***Testability*** and ***Domain-specific Attributes***. These subcharacteristics are identified based on the analysis of the software quality challenges and assessment [Fitzpatrick et al. 2004], the types of change stimuli and evolution [Chapin et al. 2001], the taxonomy of software change based on various dimensions that characterize or influence the mechanisms of change [Buckley et al. 2004], and experiences we gained in industrial case studies [Breivold and Crnkovic 2008]. Paper A outlines a software evolvability model, in which subcharacteristics of software evolvability and corresponding measuring attributes are identified. The idea with the evolvability model is to further derive the identified subcharacteristics to the extent when we are able to quantify them and/or make appropriate reasoning about the quality of the attributes. This model is established as a first step towards analyzing and quantifying evolvability, a base and check point for evolvability evaluation and improvement. Additionally, paper B describes evolvability subcharacteristics, correlating to the problems in the case of an industrial automation control system.

How can software evolvability be assessed in a systematic manner? **(Q2)**

Paper B describes our work in analyzing an industrial automation control system, driven by the need to improve its evolvability. A structured method has been proposed and piloted for analyzing evolvability at the architectural level, i.e. the ARchitecture Evolvability Analysis (AREA) method. The method consists of three phases:

Phase 1: Analyze the implications of change stimuli on software architecture. As change stimuli have impact on the software system in terms of software structures and/or functionality, this phase analyzes the impact of change stimuli on the current architecture. Phase 1 consists of the following two steps:

- **Step 1.1: Identify potential requirements in the software architecture**. The aim of this step is to extract potential requirements that are essential for software architecture to accommodate change stimuli.
- **Step 1.2: Prioritize potential requirements in the software architecture**. All the potential requirements identified from the first step need to be prioritized, in order to establish a basis for common understanding of the architecture requirements among stakeholders within the organization.

Phase 2: Analyze and prepare the software architecture to accommodate change stimuli and potential future changes. This phase focuses on the identification of potential improvement proposals for the components that need to be refactored. Phase 2 consists of the following four steps:

- **Step 2.1: Extract architectural constructs related to the respective identified requirement**. We mainly focus on architectural constructs that are related to each identified potential architectural requirement.
- **Step 2.2: Identify refactoring components for each identified requirement**. In this step, we identify the components that need refactoring in order to fulfill the prioritized requirements.
- **Step 2.3: Identify and assess potential refactoring solutions from technical and business perspectives**. Potential refactoring proposals are identified and design decisions are taken in order to fulfill the requirements derived from the first phase. The change

propagation of the effect of refactoring need to be considered, as it provides an input to the business assessment, estimating the cost and effort in refactoring work.

- **Step 2.4: Define test cases**. New test cases that cover the affected component, modules or subsystems are identified.

Phase 3: Finalize the evaluation. In this phase, the previous results are incorporated, analyzed and structured into a collection of documents.

- **Step 3.1**: **Analyze and present evaluation results**. The evaluation results include (i) the identified and prioritized potential requirements on the software architecture; (ii) the identified components/modules that need to be refactored for enhancement or adaptation; (iii) refactoring investigation documentation which describes the current situation, the new requirements, potential improvement proposals and respective rationale to each identified candidate that need to be refactored, including estimated workload; (iv) test scenarios; and (v) impact analysis on evolvability in terms of each subcharacteristic.

Through the evolvability analysis process, the implications of the potential improvement proposals and evolution path of the software architecture are analyzed with respect to the evolvability subcharacteristics. The result is that the architecture requirements, corresponding architectural decisions, rationale and architecture evolution path become more explicit, better founded and documented, and that the resulting documentation of refactoring improvement proposals are widely accepted by the involved stakeholders.

Detailed Studies

What modularization means can be used to support software architecture evolution? **(Q1.1)**

Through an industrial case study in static dependency analysis, paper C explores the relationship between software evolvability, modularity and inter-module dependency. Inter-module dependency is one of many indicators and measures for achieving modularity. One way to visualize these inter-module dependencies is through the Design Structure Matrix (DSM), which is a representation and analysis mechanism for system modeling with respect to system decomposition and integration. Paper C describes also the experiences and reflections on using dependency model to support software architecture evolution. In addition, as part of the dependency analysis process, some means for providing modularization are identified, e.g.

- Design principles
- Software engineering paradigms
- Object-oriented design patterns
- Formal specification
- Programming languages
- Modeling techniques
- Architecture styles

These means can be used to support software evolution and to provide one way to let some part of a system change independently of all other parts. An additional observation is the potential of combining different means for improved modularization and quality attributes, thus to support software evolution.

Given the technology-type change stimulus of introducing SOSE to CBSE, what impacts need to be considered? **(Q3.1)**

In order to analyze the impacts of the introduction of SOSE to CBSE, the first step is to achieve good understandings of the characteristics of and possibilities provided by the two engineering paradigms. Accordingly, taking CBSE and SOSE engineering paradigms as examples, paper D exemplifies the necessity of making analysis and exploration of both existing and emerging technologies for better understanding and utilization of both. Paper D presents a comparison framework for component-based and service-oriented software engineering from the following perspectives:

- ***Key concepts*** with respect to module, specification, interface and assembly;
- ***Key principles*** with respect to coupling, self describing, self contained, state and location transparency;
- ***Development process***;
- ***Technology concerns*** with respect to technology neutrality, encapsulation, and static vs. dynamic;
- ***Quality concerns*** e.g. reusability, substitutability and interoperability;

- ***Composition concerns*** e.g. heterogeneous vs. homogeneous composition, design time/run time composition and composition mechanisms, as wells as predictability.

In paper D, a brief discussion of reasonable utilization, combination and adaptation of the two paradigms is also outlined through looking into a set of research studies in how they have been used for improved quality attributes. The result is that as both CBSE and SOSE can co-exist in enterprise systems and complement each other [Wang and Fung 2004], a good understanding of both technologies and a thorough analysis of their impacts on quality attributes will lead to more efficient combination and adaptation of these paradigms in future software development.

In this thesis, we have only partially answered the research question Q3.1 through providing an explicit clarification of the concepts, principles and characteristics of CBSE and SOSE. This is the first necessary step before further exploration in efficient utilization and reasonable combination of CBSE and SOSE in future applications. It is also a necessary step before further investigation of the impacts of the introduction of SOSE to CBSE. However, a continuation of further investigations of the impacts of the introduction of SOSE to CBSE is not within the focus of this thesis. It remains to be one of the areas for future work (refer to chapter 5).

Given the business-type change stimulus of adopting a product line approach, what impacts need to be considered from a software evolution perspective? **(Q3.2)**

In order to analyze the impacts of the adoption of a product line approach, we performed two industrial case studies, driven by the need to transform the existing legacy systems towards product line architectures in order to improve evolvability. Paper E describes our work in these two cases and proposes a structured product line migration method with focus on the migration process when the migration decision has been made. The method consists of five steps:

- **Step 1: Identify requirements on the software architecture**. In this step, requirements essential for a cost-effective software architecture transition to product line architecture are extracted.
- **Step 2: Identify commonalities and variability**. In this step, common core assets and variability to facilitate product derivation are identified.

- **Step 3: Restructure architecture**. In this step, the product line architecture is constructed.
- **Step 4: Incorporate commonality and variability**. In this step, feasible realization mechanisms and potential improvement proposals to facilitate the revised product line architecture are defined.
- **Step 5: Evaluate software architecture quality attributes**. In this step, the impact of potential improvement proposals on the quality attributes of the product line architecture is evaluated.

In addition, applying a software product line approach to legacy systems requires that care is taken to ensure that critical aspects are considered for a smooth and successful product line migration. Through the two industrial cases, observations have been made with respect to business, organization, development process and technology perspectives when adopting a product line approach. These observations and experiences from the case studies are also described in paper E to recommend practices that are particularly useful. Some examples are:

Business perspective:

- Different triggers for decisions to adopt a product line approach exist. Business objectives motivate architecture and process changes. The triggers for these changes might appear different although the decision to have a product line approach might be the same.
- Improve risk management through constant progress measuring.

Organization perspective:

- Product managers for different products using the product line architecture should synchronize needs.
- Define roles, responsibilities and ways to share technology assets.

Process perspective:

- Perform the migration to product lines through incremental transitions.
- Ensure communication between technology core team and implementation team.

Technology perspective:

- Use tool support for dependency analysis.

- Use architecture documentation to improve architectural integrity and consistency.
- Carefully define variation points and realization mechanisms.

2.1 Summary of Thesis Contributions

The contributions of the thesis are visualized in Figure 1.

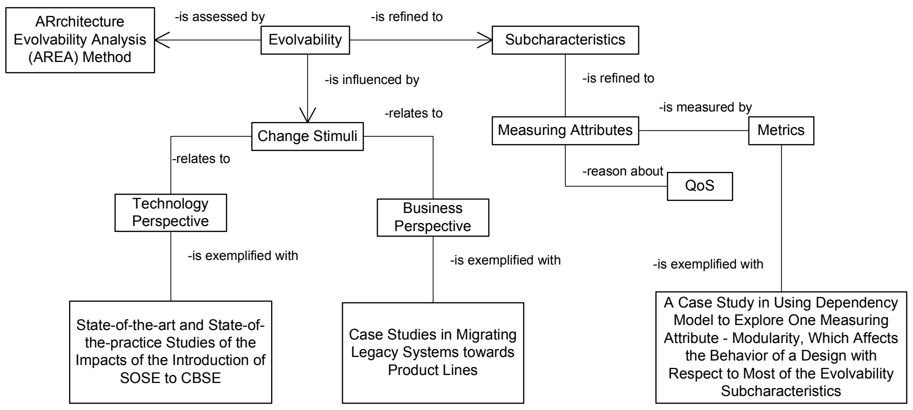

ARrchitecture Evolvability Analysis (AREA) Method
-is assessed by
Evolvability
-is refined to
Subcharacteristics
-is influenced by
-is refined to
-relates to
Change Stimuli
Measuring Attributes
-is measured by
Metrics
-relates to
-reason about
QoS
Technology Perspective
Business Perspective
-is exemplified with
-is exemplified with
-is exemplified with
State-of-the-art and State-of-the-practice Studies of the Impacts of the Introduction of SOSE to CBSE
Case Studies in Migrating Legacy Systems towards Product Lines
A Case Study in Using Dependency Model to Explore One Measuring Attribute - Modularity, Which Affects the Behavior of a Design with Respect to Most of the Evolvability Subcharacteristics

Figure 1. Contributions of the Thesis

We outline in this thesis a software evolvability model that provides a basis for analyzing and evaluating software evolvability. This model refines software evolvability into a collection of subcharacteristics that can be measured through a number of measuring attributes. Moreover, we further explore one particular measuring attribute, i.e. modularity, which affects the behavior of a design with respect to most of the evolvability subcharacteristics. This is because designing software for ease of extension and contraction depends on how well the software structure is organized, and modular designs are argued to be more evolvable, i.e. these designs facilitate making future adaptations.

We introduce a structured method for analyzing evolvability at the architectural level - the ARchitecture Evolvability Analysis (AREA) method that focuses on improving the capability in being able to on forehand understand and analyze systematically the impact of a change stimulus. The method is studied in an industrial setting.

The fact that change stimuli come from both technical and business perspectives spawns two aspects that we also focus on in the thesis, i.e. to investigate the impact of technology-type and business-type of change stimuli. For technology-type of change stimulus, we take CBSE and SOSE engineering paradigms as examples and investigate the impact of the emergence of a new engineering paradigm. We exemplify the necessity of making analysis and exploration of both existing and emerging technologies. For business-type of change stimulus, we focus on managing the migration of legacy systems towards product lines due to the need for differentiation in the marketplace, with short time-to-market as part of the need. Two industrial cases are studied in detail. Observations are made with respect to business, organization, development process and technology perspectives when adopting a product line approach. The experiences from the case studies are also described to recommend practices that are particularly useful.

Chapter 3. Research Method

This chapter includes an overview of the relevant research methods used in software engineering and how these methods are used in the research presented in this thesis. Some of the papers included in the thesis describe how a specific method is applied in that part of the research. The general research process and the overall validity of the studies are discussed here.

The ACM SIGCSE committee on teaching Computer Science Research Methods (SIGCSE-CSRM) [SIGCSE] describes a research process framework [Holz et al. 2006]. The framework consists of four different questions that as a whole describe the general research process:

- Question A: What do we want to achieve?
- Question B: Where does the data come from?
- Question C: What do we do with the data?
- Question D: Have we achieved our goal?

To answer these questions in the general research process, different research methods have been outlined [Holz et al. 2006]. Moreover, Shaw characterizes software engineering research and develops a research classification framework, which describes the kind of answers that are of interest for software engineering research, the research methods that are adopted and the criteria for evaluating the results [Shaw 2002]. She classifies research based on the type of the following three aspects:

- Research questions: What kinds of research questions are interesting for software engineering researchers? This corresponds to question A in the general research framework, i.e. what do we want to achieve?
- Research results: A classification of the kind of research results, which help to answer the research questions. This covers question C in the general research framework, i.e. what do we do with the data? This also covers question B, i.e. where does the data come from?

- Validation techniques: The framework classifies the kind of evidence that can be used to demonstrate the validity of the result. This relates to question D in the general research framework, i.e. have we achieved our goal?

The detailed descriptions of the research questions and the research results are covered in chapter 1 and chapter 2 respectively. The research process and method as well as the validity of the research results are discussed in the following sections.

3.1 Research Process and Method

The research process conducted in this thesis consists of the following steps:

1. Analysis of the state-of-the-art and state-of-the-practice of the existing software quality models (refer to section 4.2) for software evolution;
2. Analysis of the state-of-the-art and state-of-the-practice of the existing software process models (refer to section 4.3) for software evolution;
3. Case studies performed to understand subcharacteristics of the evolvability of a software system;
4. Analysis of the state-of-the-art and state-of-the-practice of component-based and service-oriented software engineering (refer to section 4.6) to investigate impacts of technology advances;
5. Case studies performed to investigate impacts of migrating legacy software systems to the product line software development (refer to section 4.7).

Through the first two steps, a thorough investigation of the well-known software quality models is made and the idea of a characterization of software architecture evolvability is outlined. Afterwards, a characterization of the evolvability of an industrial software system is studied and created in the third step. This characterization and the results from the case study are reported in paper A and B. Furthermore, paper C reports an in-depth study of one of the measuring attributes identified in the evolvability characterization. The analysis of the particular measuring attribute is performed through another industrial case study, in which the software architecture evolution is supported through the usage of dependency model. The data collection for paper D is based on literature surveys through the fourth step. The fifth step includes two case studies with two different development organizations in different domains to address the impacts of product line migration. The migration process and the results from the case studies are reported in paper E.

A summary of the computing research methods can be found in [Holz et al. 2006]. Among them, the following specific research methods are used in this thesis for data collection:

- *Interview* [Benyon et al. 2005]: This is a research method for gathering information. People are posed questions by an interviewer. The interviews may be structured or unstructured both in the questions asked by the interviewer, as well as the answers available to the interview subject. In the research presented in this thesis, we performed unstructured interviews.
- *Critical Analysis of the Literature* [Zelkowitz and Wallace 1997]: This research method is a historical method, which collects and analyzes data from published material. Literature search requires the investigator to analyze the results of papers and other documents that are publicly available. The research context and background to paper A (regarding the analysis of existing software quality models) and paper D (regarding the state-of-the-art and state-of-the-practice of CBSE and SOSE) are originated from this specific method.
- *Lessons-learned* [Zelkowitz and Wallace 1997]: Lessons-learned documents are often produced after a large industrial project is completed, whether data is collected or not. A study of these documents often reveals qualitative aspects which can be used to improve future developments. Parts of the results reported in paper C (regarding the experiences and reflections through the dependency analysis) and paper E (regarding the observations and recommendations in product line migration) are lessons-learned throughout the case study executions.
- *Qualitative Research* [Gay and Airasian 1999]: This method is the collection of extensive narrative data on many variables over an extended period of time, in a naturalistic setting, in order to gain insights not possible using other types of research. The results presented in paper B (regarding the impact analysis of potential refactoring solutions on evolvability subcharacteristics) belong to this category.

- *Quantitative Research* [Gay and Airasian 1999]: This method is the collection of numerical data in order to explain, predict and/or control phenomena of interest. The results presented in paper C (regarding the inter-module dependencies) belong to this category.
- *Case Study* [Fenton and Pfleeger 1997]: This is a research technique in which key factors that may affect the outcome of an activity are identified and the activity are documented, including its inputs, constraints, resources and outputs. Two types of case study are described in [Yin 2003]. They are:
    - Single Case: It examines a single organization, group, or system in detail; involves no variable manipulation, experimental design or controls. The results presented in paper B (regarding the software evolvability analysis) are derived from a single organization and belong to this category.
    - Multiple Case Studies: They are as for single case studies, but carried out in a small number of organizations or context. The results presented in paper E (regarding the observations and experiences gained through the product line migration process) are derived from two organizations in two different domains and belong to this category.

3.2 Validity Discussions

Based on [Yin 2003] and [Wohlin and Wesslen 2000], four types of validity are considered in this thesis: construct validity, internal validity, external validity, and reliability.

Construct validity relates to the collected data and how well the data represent the investigated phenomenon, i.e. it is about ensuring that the construction of the study actually relates to the research problem and the chosen sources of information are relevant. The *construct validity* can be increased through the following tactics [Yin 2003]:

- Use multiple sources of evidence;
- Establish chain of evidence;
- Have key informants review draft of case study report.

Internal validity concerns the connection between the observed behavior and the proposed explanation for the behavior, i.e. it is about ensuring that the actual conclusions are true. The *internal validity* is 'only a concern for causal (or explanatory) case studies' [Yin 2003]. It can be increased through the following tactics:

- Do pattern-matching;
- Do explanation-building;
- Address rival explanations;
- Use logic models.

External validity concerns the possibilities to generalize the results from a study. It can be increased through the following tactics [Yin 2003]:

- Use theory in single-case studies;
- Use replication logic in multiple-case studies.

Reliability concerns the possibilities to reach the same conclusions if the study is repeated by another researcher. It can be increased through the following tactics [Yin 2003]:

- Use case study protocol;
- Develop case study database.

Because the ways for the data collection and research design vary when we answer each research question, we go through each research question in the following subsections and describe respective type of the validation used.

3.2.1 Research Question 1: What subcharacteristics are of primary importance for the evolvability of a software system?

The *construct validity* is addressed through using multiple sources of evidence, including critical analysis of the existing literature and an industrial case study [Breivold and Crnkovic 2008]. We collect and analyze data from published materials. The criteria on which the literature is being evaluated include software evolution related areas which cover a broad range of topics, such as software quality models, software process models, software quality metrics, and software architecture evaluation. In addition, the industrial case study, though is a single-case, is a representative and typical case which captures the commonplace situation of large complex software systems.

Our case study is explorative, and hence less sensitive to the *internal validity* which is only a concern for causal (or explanatory) case studies.

The *external validity* is addressed through analytical generalizations in the case study. However, we do not exclude the possibilities that other domains or cases might have extended or different set of evolvability subcharacteristics. We cannot with certainty say that this is the case. Further studies are needed in order to draw such conclusions. For this reason we precisely defined the scope and the context of the research.

A basis for achieving *reliability* is to have a well-documented case study protocol, which is the case in the research presented in this thesis. The documentation on architectural requirements and quality improvement requirements is available. However, different people might interpret textual materials in different ways, which might lead to different set of abstractions on evolvability subcharacteristics. We address this by having the key software architect and several researchers to review the documents, e.g. software architecture requirements, and documents concerning the analysis of the case study.

3.2.2 Research Question 2: How can software evolvability be assessed in a systematic manner?

The *construct validity* is addressed through triangulation, i.e. multiple sources for the data in the project:

- Architecture workshops with stakeholders to extract potential architectural requirements; these architectural requirements are checked against the evolvability subcharacteristics for the justification of whether the realization of each requirement would lead to an improvement of the subcharacteristics (or possibly a decrease, which would then require a tradeoff decision).
- The involvement of software architects and senior software developers in the analysis process;
- The researchers' experiences and involvement in the software product development;
- Discussions with involved stakeholders on software architecture requirement documents, potential architecture improvement proposals and their respective quality impact analysis to ensure software evolvability and to avoid risks to its decrease.

Our case study is explorative, and hence less sensitive to the *internal validity* which is only a concern for causal (or explanatory) case studies.

The *external validity* is addressed through analytical generalizations in the case study, in which we perform and pilot the software evolvability analysis method. A possible consideration is whether the analysis method can be generalized to a different organization or a different domain. We assume that the analysis method can be generalized, as the method and the procedures in performing the method are not constrained by any domain or organization related factors. However, further studies are needed in order to further refine and validate the method. Another perspective with respect to the external validity is to perform new evolvability assessment case studies and compare the results, including the estimation of the efforts needed to analyze evolvability. This can be done in stages, i.e. firstly, in the same or similar domain/context, and secondly, in different contexts. This multiple case study remains to be done.

Reliability is addressed through the detailed description of the procedures used in the analysis method, proper documentation of the results in each performed step in the case study, as well as reviews of the software architecture requirement documents and the potential architecture improvement proposals by the involved software architects, senior software developers and researchers.

3.2.3 Research Question 1.1: What modularization means can be used to support software architecture evolution?

The *construct validity* is addressed through triangulation. One of the means applied in the case study is using dependency model to support software architecture evolution. The idea is to use inter-module dependency as one of many indicators and measures for achieving modularity. A subset of the complete software system is analyzed through using inter-module dependency to measure its modularity. The modularization is performed through simulating changes in the dependency model without of making any modifications to the actual source code. Afterwards, the resulting modularity is compared with the previous one before the simulated changes.

Our case study is explorative, and hence less sensitive to the *internal validity* which is only a concern for causal (or explanatory) case studies.

The *external validity* is addressed through analytical generalizations in the case study. The purpose of the analysis in the case study is to visualize dependencies to provide indications to the hotspots in the software architecture and software implementation, thus to support the software architecture evolution. The conclusion of using dependency model to support software architecture evolution can be generalized, as the inter-module dependency is an objectively quantitative indicator.

Reliability is addressed through the detailed description of the procedures performed in the dependency analysis process, proper documentation of the resulting dependency model from each step in the case study, as well as reviews of the software architecture improvement proposals by the stakeholders and researchers. Our software evolution experiences with respect to the reflections from the dependency analysis process are gained through:

- The daily meetings with the stakeholders, e.g. the software architect and senior software developers to discuss the progress and the solutions to any encountered problems;
- The researchers' experiences and involvement in the software product development;
- The reviewing of software architecture analysis documents and potential improvement proposals to ensure that the collected data is relevant.

3.2.4 Research Question 3.1: Given the technology-type change stimulus of introducing SOSE to CBSE, what impacts need to be considered?

The *construct validity* is addressed through critical analysis of the existing literature with regard to component-based and service-oriented software engineering, as well as through the reviews from several researchers in these areas. We collect and analyze data from published materials [Crnkovic and Larsson 2002; Stojanovic and Dahanayake 2005] and other related publications. The criteria on which the literature is being evaluated include component-based and service-oriented software engineering related areas as well as their utilizations.

Our case study is explorative, and hence less sensitive to the *internal validity* which is only a concern for causal (or explanatory) case studies.

The *external validity* is addressed through analytical generalizations from the evaluated literatures. We introduce the comparison framework between CBSE and SOSE, through characterizing the key concepts, key principles, quality concerns, composition mechanisms, utilization and combination of both technologies. The conclusion of the paper is 'a good understanding of both technologies and a thorough analysis of their impacts on quality attributes will lead to more efficient combination and adaptation of these paradigms in future software development'. This conclusion is based on the comparison framework and related works that describe how the two technologies have been combined for improved quality attributes. We assume that the conclusion from the analysis can be generalized with any technology-type of change stimuli due to the abstraction level.

Reliability is addressed through well-structured data collection from the literatures. However, different people might interpret textual materials in different ways, which might lead to different set of abstractions and slightly different comparison framework. We address this by having several researchers to review the proposed comparison framework.

3.2.5 Research Question 3.2: Given the business-type change stimulus of adopting a product line approach, what impacts need to be considered from a software evolution perspective?

The *construct validity* is addressed through triangulation. The reported migration experiences and observations are gained through multiple sources for the data in the project:

- Analysis of two different industrial software systems from two different domains;
- Analysis of two different organization structures with distributed development teams;
- The involvement of the stakeholders of different roles (e.g. product management, software architects and senior software developers) for each case study;
- The researchers' experiences and involvement in the software product development to ensure that the collected data is relevant;
- Regular meetings and workshops for open discussions.

Our case study is explorative, and hence less sensitive to the *internal validity* which is only a concern for causal (or explanatory) case studies.

The *external validity* is addressed through the selection of studied systems from two different domains, including automation control system, power protection and control system. Besides, external validity is also addressed through the selection of different organizations with different organization structures. The product line development is organized in two ways: (i) in a separate product line team – one team develops the core assets while other teams develop products; or (ii) within the product team – the development team is responsible for both product and core asset development. Both organization structures are reflected in the two case studies.

Reliability is addressed through the detailed description of the procedures used in the product line migration process, proper documentation of the results from each performed step in the case study, as well as reviews of these documents by the stakeholders and researchers. However, different people might interpret textual materials in different ways, which might lead to slightly different set of observations and experiences. We address this by having several researchers to review the experience analysis extracted from the case studies.

Chapter 4. Related Work

This chapter relates the work in this thesis to relevant research and practice areas, subdivided into a number of sections. In each section, there is also an explanation of how the thesis is related to each area.

Section 1 presents a brief overview of the observed behavior of software systems and challenges encountered during software evolution. Section 2 provides a survey of the existing well-known software quality models, which form the basis for the establishment of our evolvability model. Section 3 surveys the software process models as software architecture evolution is inseparably bound to a process context, e.g. the need to cost-effectively carry out software evolution during the software system's lifecycle. Section 4 briefly describes software architecture evolution with regard to its qualitative and quantitative assessment as well as the architectural integrity issue which is one of the aspects that we take into consideration during evolvability analysis. Section 5 presents an overview of software architecture evaluation methods. Good understanding of their applicability and limitations is the basis for the proposed software architecture evolvability analysis method in this thesis. Section 6 presents a brief overview of component-based and service-oriented software engineering, as one of the detailed research questions that we try to answer in this thesis is closely related to this area. Section 7 describes briefly the software product line engineering methods and process, which are of close relevance as one of our detailed research questions deals with the adoption of a product line approach. Section 8 describes reverse engineering and reengineering, and section 9 describes briefly software quality metrics that are related to software evolution.

4.1 Software Evolution

The laws of software evolution is formulated in [Lehman 1980; Lehman et al. 1997], based on the observations of the IBM OS/360 operating system and the FEAST project. The term software evolution is deliberately used in Lehman's work to address the difference with the post-deployment activity of software maintenance. He uses the term E-type software to denote programs that must be evolved because they operate in or address a problem or activity of the real world. Accordingly, changes in the real world will affect the software and require subsequent adaptations.

The laws of software evolution encapsulate observed behavior of a number of evolving systems over the years and are summarized as follows:

- *Continuing change* An E-type system that is used must be continually adapted else it becomes progressively less satisfactory.
- *Increasing complexity* As an E-type system evolves its complexity increases unless work is done to maintain or reduce it.
- *Self regulation* Global E-type system evolution processes are self regulating.
- *Conservation of organizational stability* Average global activity rate in an E-type process tends to remain constant over periods or segments of system evolution.
- *Conservation of familiarity* The average growth rate of E-type systems tends to remain constant or to decline.
- *Continuing growth* The functional capability of an E-type system must be continually increased to maintain user satisfaction over its lifetime.
- *Declining quality* Unless rigorously adapted to take into account changes in the operational environment, the quality of E-type systems will appear to be declining.
- *Feedback system* E-type software processes are multilevel, multi-loop, multi-agent feedback systems.

The software architecture is inevitably subject to evolution due to the above-mentioned phenomena of software evolution, for instance continuing change, increasing complexity, continuing growth and declining quality.

Additionally, the following properties of large software systems are noted in [Brooks 1987].

- *Complexity* An essential property of large software systems, leading to the following problems:

    - Difficulty of communication among development team members, leading to product flaws, cost overruns and schedule delays;
    - Difficulty of understanding all the possible states of the program;
    - Difficulty of extending programs to new functions without creating side effects;
    - Difficulty of getting an overview of the system, thus impeding conceptual integrity.
- *Conformity* Many software systems are constrained by the need to conform to human institutions and systems.
- *Changeability* The software entity is constantly subject to pressures for change.
- *Invisibility* Software is invisible and unvisualizable. There is no geometric representation. Instead, there are several distinct but interacting graphs of links that represent different aspects of the system.

The properties of large software systems noted in [Brooks 1987], e.g. software complexity, inevitable changes of software systems and invisibility in terms of software structure representation, further confirm the software evolution phenomena and exhibit the intensified need on having evolvable software systems that accommodate changes in a cost-effective way while maintaining the architectural integrity. Without active countermeasures, the quality of a software system will gradually degrade as the system evolves.

Moreover, software aging is inevitable. Parnas uses the metaphor of decay to describe how and why software becomes increasingly brittle over time [Parnas 1994]. There are two types of software aging which can lead to rapid decline in the value of a software product. The first is caused by the failure of the product's owners to modify it to meet changing needs; the second is the result of the changes that are made. Both types of software aging in turn lead to inadequate evolvability. Following problems are associated with software aging [Parnas 1994]:

- Inability to keep up with the market due to increasing size and complexity;
- Reduced performance due to the gradually deteriorating structure;

- Decreased reliability because of errors introduced when changes are made.

4.1.1 Relation to the Thesis

In order to keep the system useful as it was, we must continually adapt it to the ever-changing requirements. This exhibits the need on having an evolvable software system. Therefore, the software evolution retraces motivate the reasons for the thesis, i.e. we need to investigate means to analyze, characterize and measure software evolvability.

4.2 Software Quality Models

A quality model provides a framework for quality assessment. It aims at describing complex quality criteria through breaking them down into concrete subcharacteristics. A general description of different quality models can be found in [Ortega et al. 2003]. In quality models, quality attributes are decomposed into various factors, leading to various quality factor hierarchies. Some well-known quality models are McCall's quality model [McCall et al. 1977], Dromey's quality model [Dromey 1996], Boehm's quality model [Boehm et al. 1978], ISO 9126 [ISO9126] and FURPS quality model [Grady and Caswell 1987].

4.2.1 McCall's Quality Model

McCall's quality model [McCall et al. 1977] addresses three perspectives for defining and identifying the quality of a software product:

- *Product operation* is the product's ability to be quickly understood, operated and capable of providing the results required by the user. It covers modifiability, reliability, efficiency, integrity and usability.
- *Product revision* is the ability to undergo changes. It covers maintainability, flexibility and testability.
- *Product transition* is the adaptability to new environments. It covers portability, reusability and interoperability.

This model further details the above three perspectives into a hierarchy of factors, criteria and metrics.

4.2.2 Boehm’s Quality Model

Boehm’s quality model [Boehm et al. 1978] begins with the software’s general utility, i.e. the high level characteristics that represent basic high-level requirements of actual use. The general utility is refined into:

- *Portability*
- *Utility* It is further refined into reliability, efficiency and human engineering.
- *Maintainability* It is further refined into testability, understandability and modifiability.

Boehm’s quality model is similar to McCall’s quality model in that it represents a hierarchical structure of characteristics, each of which contributes to the total quality.

4.2.3 FURPS Quality Model

FURPS [Grady and Caswell 1987] stands for functionality, usability, reliability, performance and supportability. Two steps are considered in this model: setting priorities and defining quality attributes that can be measured.

4.2.4 ISO 9126 Quality Model

ISO 9126 [ISO9126] specifies and evaluates the quality of a software product from different perspectives. Product quality is defined as a set of product characteristics. The characteristics that are observed by the end-user on the final software product are called external quality characteristics. The characteristics that relate to software development process and environment or context are called internal quality characteristics. An external characteristic can be measured internally, and is determined or influenced by the internal characteristics. The model categorizes software quality attributes into six characteristics: functionality, reliability, usability, efficiency, maintainability and portability. One advantage of this quality model is that it defines the internal and external quality characteristics of a software product.

4.2.5 Dromey’s Quality Model

[Dromey 1996] proposes a working framework for evaluating requirement determination, design and implementation phases. Corresponding to the products resulted from each stage of the development process; the framework consists of three models:

- *Requirement model* The high-level attributes for the requirement quality model are accurate, understandable, implementable, adaptable, and process mature.
- *Design model* The high-level attributes for the design quality model include accurate; effective, understandable, adaptable and process mature.
- *Implementation quality model*

The information extracted from each model can be used to build, compare and evaluate the quality of a software product. In Dromey's quality model, process maturity is an aspect that has not been considered in previous models.

4.2.6 Relation to the Thesis

The quality characteristics that are addressed in these quality models are summarized in Table 1. As shown in Table 1, the term evolvability or similar is not explicitly used in either of the quality models. Nevertheless, several quality attributes are correlated to software evolvability, e.g. adaptability, extensibility and maintainability. However, based on the definition of evolvability in [Rowe et al. 1994], the multifaceted quality attribute software evolvability covers more aspects than adaptability, extensibility or maintainability. Through analyzing the software quality challenges and assessment [Fitzpatrick et al. 2004], the types of change stimuli and evolution [Chapin et al. 2001], the taxonomy of software change based on various dimensions that characterize or influence the mechanisms of change [Buckley et al. 2004], and experiences we gained in industrial case studies [Breivold and Crnkovic 2008], we have discovered that only having a collection of the subcharacteristics of maintainability as defined in the ISO software quality standard [ISO9126] is not sufficient for a software system to be evolvable. This poses one of the goals for our research, i.e. to investigate characteristics that are of primary importance for the evolvability of a software system, and to outline a software evolvability model that provides a basis for analyzing and evaluating software evolvability.

Table 1. Quality Characteristics Addressed in Quality Models

Quality Characteristics	McCall	Boehm	FURPS	ISO 9126	Dromey
Adaptability			x	x	
Compatibility			x		
Correctness	x				
Efficiency	x	x		x	x
Extensibility			x		
Flexibility	x				
Human Engineering		x			
Integrity	x				
Interoperability	x			x	
Maintainability	x	x	x	x	x
Modifiability		x		x	
Performance			x		
Portability	x	x		x	x
Reliability	x	x	x	x	x
Reusability	x				x
Supportability			x		
Testability	x	x		x	
Understandability		x		x	
Usability	x		x	x	x

4.3 Software Process Models

The primary functions of a software process model are to determine the order of the stages involved in software development and evolution, and to establish the transition criteria for progressing from one stage to the next [Boehm 1988]. Several process models have been proposed and gained widespread acceptance since the late seventies as the term software evolution was deliberately used and recognized by the research community. Below is an overview of the process models, with focus on those models that take constant changes and software evolution into consideration.

4.3.1 Waterfall Model

[Royce 1987] proposes the waterfall lifecycle process for software development. In this process, several stages are described as taking place in sequence, i.e. requirement analysis, design, implementation, testing and maintenance. In this process model, there is no iteration in the process. Although the waterfall model's approach helps eliminate many difficulties previously encountered in software projects, the inherent limitations of this software process model are that the separation in phases is too strict and inflexible, and that it is often unrealistic to assume that the requirements are known before starting the software design phase. The emphasis on fully elaborated documents as completion criteria for early requirements and design phases creates a primary source of difficulty when the requirements continue to change during the entire software life cycle as in many cases. Moreover, in this process model, the maintenance phase is the final phase of a software system's lifecycle. Only bug fixes and minor adjustments to the software are performed during this phase. Therefore, the maintenance stage needs to be expanded to represent broader activities, i.e. not only maintaining the originally designed functions, but also adding new functions, coping with changing environments and changing requirements.

4.3.2 Change Mini-Cycle Process Model

[Yau et al. 1978] proposes a process model with the so called change mini-cycle, in which change impact analysis and change propagation are identified to accommodate the fact that software changes are rarely isolated. In this process model, software evolution is described in terms of the change mini-cycle, which consists of several phases:

- *Change request*;
- *Change planning* includes:

    - Software comprehension to understand what parts of the software will be affected by a requested change;
    - Change impact analysis to predict the parts that are likely to be affected by a change.
- *Change implementation* includes:
    - Restructuring for change to improve the software structure or architecture without changing the behavior;
    - Change location;
    - Propagation of change due to the non-local impact nature of a change.
- *Validation of change*

The assumptions of the proposed process model are that the requirements continue to change during the entire lifetime of a software project, and that the knowledge gained during the later phases may become feedbacks to the earlier phases.

4.3.3 Evolutionary Development Model

Gilb proposed an "evolutionary development model", in which the key word is incremental delivery, implying real deliveries to a real user. According to [Gilb 1981], "You must evolve in small steps towards your goals; large step failure kills the entire effort. And early frequent result delivery is politically and economically wise. 2% of total is a small step that you can afford to fail on."

The assumption of this model is that the software engineering is, by nature, playing with the unknown [Gilb 2002]. One way to deal with these many unknowns is to tackle them in small increments, one at a time. These small increments are not mere development increments. It is important to note that they are incremental satisfaction of identified stakeholder requirements.

4.3.4 Spiral Model

The spiral model [Boehm 1988] proposed by Boehm is a risk-driven approach to the software process rather than a primarily document-driven approach such as the waterfall model or code-driven process such as the evolutionary development. A typical cycle of the spiral consists of the following steps:

- Identification of the objectives of the portion of the product being elaborated, alternative means of implementing this portion of the

product, and the constraints imposed on the application of the alternatives;
- Evaluation of the alternatives relative to the objectives and constraints to identify risks;
- Risk resolution;
- Development and verification of next level product.

In this process model, prototyping is incorporated as a risk reduction option at any stage of development. In addition, the model accommodates reworks or go-backs to earlier stages as new alternatives are identified or as new risk issues need resolution.

4.3.5 Staged Model

[Bennett and Rajlich 2000] explicitly takes into account the issue of software aging [Parnas 1994] and proposes the staged model which represents the software lifecycle as a sequence of the following stages:

- *Initial development* develops the first version of the software system to ensure that subsequent evolution can be achieved easily;
- *Evolution stage* implements any kind of modification to the software system;
- *Servicing stage* implements and tests tactical changes to the software through applying small patches to keep the software up and running;
- *Phase out* and *close down stages* manage the software towards the end of its life.

In this model, during the initial development, the main need is to ensure that the subsequent evolution can be achieved easily. During the evolution stage, the software architecture evolution is essential to respond to unexpected new user requirements. Meanwhile, we need to extend and adapt functional and nonfunctional behavior without destroying the integrity of the architecture.

4.3.6 Agile Software Development

Agile software development [Cockburn 2002; Martin 2003] is a lightweight iterative and incremental approach to software development, which is performed in a collaborative manner and explicitly needs to accommodate the changing needs of various stakeholders. The introduction of Extreme Programming [Beck 1999] is widely acknowledged as the starting point for various agile software development methods, such as Scrum [Schwaber and Beedle 2001], Feature Driven Development [Palmer and Felsing 2002], Dynamic Systems Development Method [Stapleton 1999], Adaptive Software Development [Highsmith 2000] and Open Source Software Development [O'Reilly 1999]. These methods attempt to produce working software at frequent intervals, minimize the comprehensive documentation at an appropriate level. A key aspect in these methods is responding to change, i.e. the development group, comprising both software developers and customer representatives, should consider possible adjustment needs that emerge during the development process lifecycle, and should be prepared to make changes. Changing environment in software business affects the software development processes [Highsmith and Cockburn 2001]. This requires better handling of inevitable changes throughout the project lifecycle, instead of trying to stop change early.

4.3.7 Evolution and Maintenance Management Model

SYSLAB, the Information Systems Laboratory (http://syslab.dsv.su.se/) is in the process of developing a comprehensive process model for industrial evolution and maintenance, and thus, not much data has been published yet. The model is called Evolution and Maintenance Management Model. It consists of the following models:

- Process Models within Corrective Maintenance (CM3)
  - *Front-End Problem Management* is a detailed problem management process model that is utilized at the front-end support level;
  - *Back-End Problem Management* is a detailed problem management process model that is utilized at the back-end support level;
  - *Emergency Problem Management* attends severe emergency problems that present immediate danger to people, environment, resource, general welfare or businesses.
- Process Models within Evolution (EM3)
  - *Education and Training*;
  - *Pre-delivery/Prerelease Maintenance*;
  - *Release Management.*

4.3.8 Relation to the Thesis

The objective of a software process model is to reduce cost, effort and time-to-market, to increase productivity and reliability, and to support better quality and more evolvable software [Mens and Demeyer 2008]. A good understanding of the existing software process models is necessary for us to obtain insights in how the software changes are integrated in the software development lifecycle.

In this thesis, we explore the pragmatic aspects of software evolution, i.e. the methods and tools that provide the means to analyze and control the software evolution, with focus on the existing software systems. For instance, the evolvability analysis method proposed in this thesis is applied on an existing software system. Considering the complete software lifecycle, there is also the need to apply the analysis method in the early design phase of a new development effort (refer to Chapter 5).

We acknowledge changes as an essential part of software development. We also adopt the iterative and incremental change support in, for instance, the product line migration process (refer to Chapter 2).

4.4 Software Architecture Evolution

Software architectures model the structure and behavior of a system; and present a high level view of a system, including the software elements and the relationships between them. Software architectures are inevitably subject to evolution and they can expose the dimensions along which a system is expected to evolve [Garlan 2000] and provide basis for software evolution [Medvidovic et al. 1998].

Software systems undergo two main kinds of evolution [Mens and Demeyer 2008], i.e. internal evolution and external evolution. The thesis deals with the external evolution.

- *Internal evolution* models the changes in the topology of the components and interactions as they are created or destroyed during execution. It captures the dynamics of the system.
- *External evolution* models the changes in the specification of the components and interactions that are required to cope with new stakeholder requirements. It entails adaptation of the software architecture.

There exist several approaches in describing and evolving software architecture. [Aoyama 2002] proposes cost metrics of change operation for software architecture evolution and discusses the proposed metrics in continuous and discontinuous software evolution, which are the evolution patterns observed from the evolution of several software systems. Discontinuous evolution emerges between certain periods of successive continuous evolution.

[Lung et al. 1997] describes a scenario-based approach which captures and assesses software architectures for evolution and reuse. The approach consists of a framework for modeling various types of relevant information and a set of architectural views for reengineering, analyzing, and comparing software architectures. This framework is used to model several types of information, i.e.

- *Stakeholder information* describes stakeholders' objectives, which provide boundaries for analysis;
- *Architecture information* refers to design principles or architectural objectives;
- *Quality information* refers to non-functional attributes;
- *Scenarios* describe the use cases of the system to capture the system's functionality. Scenarios that are not directly supported by the current system can be used to detect possible flaws or to assess the architecture's support for potential enhancements. Scenarios are derived from the stakeholder objectives, architectural objectives, and desired system quality attributes or objectives.

The software architecture of an evolvable software system should allow changes in the software and evolve in a controlled way without compromising system integrity and invariants [Bennett and Rajlich 2000]. However, software architecture evolution often implies integrating crosscutting concerns. Therefore, architectural integrity is one aspect that needs to be taken into consideration. Otherwise, these crosscutting concerns might, if not handled with care, introduce inconsistencies and lead to evolvability degradation in the long run. To address this inconsistency issue, [Barais et al. 2004] describes a framework named TranSAT. The framework uses architectural aspect to describe new concerns and their integration into the existing architecture. The framework allows the software architect to design software architecture stepwise in terms of aspects at the design stage.

According to [Jansen and Bosch 2004], an architectural design decision is a key concept in software architecture evolution. Capturing design decisions is therefore essential to address architectural knowledge [Lago et al. 2008] vaporation issue. Otherwise, the knowledge of the design decisions that lead to the architecture is lost. Moreover, changes to the software architecture might cause violation of earlier design decisions, resulting in increased design erosion [van Gurp and Bosch 2002].

4.4.1 Relation to the Thesis

Knowledge about the implications of the software architecture evolution ensures a good understanding of the research context, for instance, we focus on external evolution in this thesis. Understanding software architecture evolution also provides us the input and background to evolvability subcharacteristics identification. For example, the architectural integrity is one aspect that needs to be considered throughout the software architecture evolution.

4.5 Software Architecture Evaluation

The foundation for any software system is its architecture, which allows or precludes nearly all of the quality attributes of the system [Clements et al. 2002]. Accordingly, several architecture evaluation methods have emerged for various purposes, e.g. to compare and identify the strengths and weaknesses in different architecture alternatives, to identify any architectural drift and erosion. Experiences of using various assessment techniques for software architecture evaluation are presented in [Christian 2006], in which scenario-based assessment, software performance assessment and experience-based assessment are addressed. A general description of different architecture analysis methods can be found in [Babar et al. 2004; Dobrica and Niemela 2002].

The following subsections describe briefly four main categories of the software architecture evaluation methods [Mattsson et al. 2006].

4.5.1 Experience-Based

Experience-based architecture evaluation means that the evaluations are based on the previous experiences and domain knowledge of developers or consultants [Avritzer and Weyuker 1999]. Some examples are:

- *Empirically-Based Architecture Evaluation (EBAE)* [Lindvall et al. 2003] defines a process for defining and using a number of architectural metrics to evaluate and compare different versions of architectures in terms of maintainability. The main steps include (i) select a perspective for the evaluation; (ii) define and select metrics; (iii) collect metrics; and (iv) evaluate and compare the architectures.

- *Attribute-Based Architectural Style (ABAS)* [Klein et al. 1999] builds on architectural styles by explicitly associating with reasoning frameworks, which are based on quality-attribute-specific models. ABAS consists of four parts: (i) *problem description* explains the problem being solved by the software structure; (ii) *stimuli and response* correspond to the condition affecting the system and measurement of the activity as a result of the stimuli; (iii) *architectural styles* are descriptions of patterns of component interaction; and (iv) *analysis* constitutes a quality-attribute-specific model that provides a method for reasoning about the behavior of interacting components in the pattern. Examples of these quality-attribute-specific models are modifiability model, reliability model and performance model.

4.5.2 Simulation-Based

Simulation-based architecture evaluation means that the evaluations are based on a high-level implementation of some or all of the components in the software architecture [Mattsson et al. 2006]. Some examples are:

- *SAM* [Wang et al. 1999] is a formal systematic methodology for software architecture specification and analysis. It is mainly targeted for analyzing the correctness and performance of a software system.
- *Argus-I* [Vieira et al. 2000] is a specification-based evaluation method that evaluates performance, dependence and correctness of a software architecture. It is also used to evaluate an architecture design with respect to structural analysis, static and dynamic behavioral analysis, model checking and simulation of architecture.

4.5.3 Mathematical Modeling

Mathematical modeling means that mathematical proofs and methods are used to evaluate operational quality requirements such as performance and reliability [Reussner et al. 2003] of the components in the software architecture. Some examples are:

- *Software Performance Engineering (SPE)* [Williams and Smith 1998] is a method for building performance into software systems. It can be used to evaluate various performance measures, e.g. response times, throughput, resource utilization and bottleneck identification.

- *Layered Queuing Networks (LQN)* [Petriu et al. 2000] is often used to evaluate the performance of a software architecture or a software system. The layered queuing network model describes the interactions between components in the architecture and required processing times for each interaction.

4.5.4 Scenario-Based

Scenario-based architecture evaluation means that quality attributes are evaluated by creating scenario profiles that force a concrete description of a quality requirement [Mattsson et al. 2006]. Some examples are:

- *Software Architecture Analysis Method (SAAM)* [Kazman et al. 1994] is originally created for evaluating modifiability of software architecture although it has been used for other set of quality attributes as well, such as portability and extensibility. The main outputs from a SAAM evaluation include a mapping between the architecture and the scenarios that represent possible future changes to the system, providing indications of potential future complexity parts in the software and estimated amount of work related to the changes.
- *Architecture Trade-off Analysis Method (ATAM)* [Clements et al. 2002] is a method for evaluating software architectures in terms of quality attribute requirements. It is used to expose the risks, non-risks, sensitivity points and trade-off points in the software architecture. It aims at different quality attributes and supports evaluation of new types of quality attributes.
- *Architecture Level Modifiability Analysis (ALMA)* [Bengtsson et al. 2004] is a method for analyzing modifiability based on scenarios. It consists of five steps: (i) set the analysis goal; (ii) describe the software architecture; (iii) elicit change scenarios; (iv) evaluate change scenarios; and (v) interpret the results. The outputs from an ALMA evaluation include: (i) maintenance prediction to estimate the required effort for system modification to accommodate future changes; (ii) risk assessment to identify the types of changes that the system shows inability to adapt to; and (iii) software architecture comparison for optimal candidate architecture.

4.5.5 Relation to the Thesis

A survey of architecture evaluation methods presented in [Mattsson et al. 2006] indicates that most evaluation methods only address one quality attribute, and very few can evaluate several quality attributes simultaneously in the same method. The survey indicates also that no specific methods evaluate testability or portability explicitly. These quality attributes can be addressed by the evaluation methods that are more general in their nature, e.g. ATAM, SAAM and EBAE. However, to analyze software evolvability which is a multifaceted quality attribute, the scenario-based methods such as ATAM would require quite a number of evolvability scenarios (to address and cover each of the seven evolvability subcharacteristics identified in our research); a more important limitation is that while scenarios are concrete anticipated events in the system lifetime, evolvability might concern high-level business requirements at an abstract level which calls for some more general type of analysis to identify the implications on software architecture and corresponding evolution path. This poses one of the motivations for our research to investigate the means to assess software architecture evolvability.

4.6 Component-Based and Service-Oriented Software Engineering

Component-based software engineering (CBSE) provides support for building systems through the composition and assembly of software components. It is an established approach in many engineering domains, such as distributed and web based systems, desktop and graphical applications and recently in embedded systems domains. CBSE technologies facilitate effective management of complexity, significantly increase reusability and shorten time-to-market.

While CBSE is an established approach in many engineering domains, the growing demands for Internet computing and emerging network-based business applications and systems are the driving forces for the emergence of service-oriented software engineering (SOSE). SOSE has evolved from CBSE frameworks and object oriented computing to face the challenges of open environments. SOSE utilizes services as fundamental elements for developing applications and software solutions. SOSE technologies offer feasibility in integrating distributed systems that are built on various platforms and technologies, and further push focus on reusability and development efficiency.

Because of the diverse nature of software systems, it is unlikely that systems will be developed using a purely service-oriented or component-based approach [Kotonya et al. 2004]. Therefore, the ability to combine the strengths of CBSE and SOSE, and use them in a complementary manner becomes essential. So far, some research has been done in combining the strengths of CBSE and SOSE for improved quality attributes of software solutions. [Jiang and Willey 2005] proposes a multi-tiered architecture that offers flexible and scalable solutions to the design and integration of large and distributed systems. The architecture makes use of both services and components as architectural elements, offering flexibility and scalability in large distributed systems and meanwhile remaining the system performance. [Wang and Fung 2004] proposes an idea of organizing enterprise functions as services and implementing them as component-based systems in order to offer flexible, extensible and value-added services. [Cervantes and Hall 2004] introduces service-oriented concepts into component models to provide support for late binding and dynamic component availability in the component models. [O'Brien et al. 2007] explores how service oriented architecture impacts a number of quality attributes, identifies issues and tradeoffs related to them. The investigated quality attributes are interoperability, performance, security, reliability, availability, modifiability, testability, usability and scalability.

4.6.1 Relation to the Thesis

Designing and implementing a large scale and complex system is a challenging task. In this thesis, we focus on two of the most well recognized software engineering paradigms that cope with this challenge, i.e. component-based software engineering (CBSE) and service-oriented software engineering (SOSE). One of the detailed research questions that we intend to address in this thesis is, by taking CBSE and SOSE as an example, to analyze the technology-type of change stimulus.

4.7 Software Product Line Engineering

A software product line is defined as "*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*" [Clements and Northrop 2002]. Product line software engineering aims to reduce cost, time-to-market, increase productivity and quality through leveraging reuse of artifacts and processes for similar products in a particular domain [Pohl et al. 2005]. It has become one of the most established strategies for achieving large-scale software reuse [Estublier and Vega 2005].

4.7.1 Software Product Line Methods

Within the area of software product line evolution, [Bosch 2000] proposes methods for designing software architecture, in particular product line architecture. [Pohl et al. 2005] elaborates two key principles behind software product line engineering: (i) separation of software development in domain and application engineering, and (ii) explicit definition and management of variability of the product line across all development artifacts. A four-dimensional software product family engineering evaluation model is described in [van der Linden et al. 2004] to determine the status of software family engineering, concerning business, architecture, organization and process.

[Faust and Verhoef 2003] presents metrics for genericity relayering, and migrates multiple instances of a single information system to a product line. [Bayer et al. 1999] presents the RE_MODEL method to integrate reengineering and product line activities to achieve a transition into product line architecture. A key element in the method is the *blackboard*, a work space which is shared for both activities that are done in parallel. The PuLSE^{TM} method [Schmid et al. 2005] addresses the different phases of product line development, and is used to systematically analyze a component and to improve its reusability as well as maintainability. The focus is on one component enabling reuse of that component. In order to evaluate the potential of creating a product line from existing products, MAP (Mining Architectures for Product Lines) [Stoermer and O'Brien 2001] focuses on the feasibility evaluation process of the organization's decision to move towards a product line. Options Analysis for Reengineering [Smith et al. 2002] is another method for mining existing components for a product line. [Maccari and Riva 2002] describes combining reference architecture and configuration architecture to describe legacy product family architecture and manage its evolution.

Research is also done in domain analysis methods. Some examples of the widely used domain analysis techniques are Feature-Oriented Domain Analysis (FODA) [Kang et al. 1990] and Feature-Oriented Reuse Method

(FORM) [Kang et al. 1998] through using feature models, in which system features are organized into trees of nodes that represent the commonality and variability within a software product line. Another notation is the orthogonal variability model [Bachmann et al. 2004; Pohl et al. 2005], which is a graph of variation points and variants.

4.7.2 Software Product Line Evolution

The ever-changing customer requirements, technology advances and internal enhancements lead to the continuous evolution of a product line's reusable assets. According to [Dhungana et al. 2008], product line evolution occurs in two dimensions as both the meta-model and the variability models can evolve independently:

- Meta-models evolve due to changes in the scope of the product line; e.g., new asset types are introduced or the product line itself is extended to support new business units.
- Variability models are subject to change whenever the product line changes; e.g., as a result of improving or extending functionality, changing technology or reorganization.

Explicit architectural knowledge is important in software evolution [Jansen 2008]. [Dhungana et al. 2006] confirms this and reports the experience of the necessity to capture architectural knowledge and make this knowledge available appropriately to various stakeholders in the product line environment. The authors argue that the architectural knowledge need to be captured by combining both top-down and bottom-up knowledge elicitation for a software product line infrastructure.

4.7.3 Product Line Engineering Process

According to [Pohl et al. 2005], the product line engineering process is composed of two sub-processes:

- *Domain engineering*: The goals of domain engineering are to define the commonality and the variability of the software product line, to define the scope of the software product line, define and construct reusable artefacts that accomplish the desired variability. The domain engineering process consists of the following five activities:
    - *Product management* defines the scope of the product line, i.e. a product roadmap that determines the major common and variable features of future products, as well as a schedule with their planned release dates. A list of the existing products and

    the development artefacts that can be reused for establishing the common platform is also defined;
    - *Domain requirement engineering* elicitates and documents the common and variable requirements for all foreseeable applications of the product line;
    - *Domain design* defines the reference architecture and a refined variability model of the product line;
    - *Domain realization* produces the detailed design and the implementation of reusable software components;
    - *Domain testing* aims to validate and verify the reusable components.
- *Application engineering*: The goals of application engineering are to achieve reuse of the domain assets, to exploit the commonality and variability of the software product line during the development of a product line application, to document the application artefacts. The application engineering process consists of the following four activities:
    - *Application requirements engineering* develops requirements specification for the particular application;
    - *Application design* produces a specialization of reference architecture for the particular application;
    - *Application realization* creates a running application with detailed design artefacts;
    - *Application testing* aims to validate and verify an application against its specification.

4.7.4 Relation to the Thesis

Product line development seldom starts from scratch. Instead, it is very often based on the existing legacy implementations [Kotonya and Hutchinson 2008]. Accordingly, a specific type of software evolution is the adoption of a product line approach and migrate existing software systems towards product line architectures. Applying a software product line approach to legacy systems requires that care is taken to ensure that critical aspects are considered for a smooth and successful product line migration. In our research, observations are made with regard to business, organization, development process and technology perspectives when adopting a product line approach. This classification has similar dimensions as in [van der

Linden et al. 2004] though we compliment with more experiences and practices.

One of the research contributions in this thesis is the proposed product line migration method with focus on the migration process when the migration decision has been made. This differs with PuLSE^{TM} method [Schmid et al. 2005] which addresses the different phases of product line development. Additionally, instead of using FODA method [Kang et al. 1990] for domain engineering, we applied product modeling in our method. The idea of constructing a federated architecture to migrate multiple instances of a single information system to a product line described in [Faust and Verhoef 2003] is similar to the way that we have performed in our case studies.

4.8 Reverse Engineering and Reengineering

Reverse engineering [Chikofsky and Cross 1990] is an important activity within software evolution. It aims at understanding the architecture or behavior of a software system through recovering and recording high-level information of a software system. The information represents abstractions that include the system structure in terms of its components and their interrelationships, the dynamic behavior of the system, functionality, modules, documentation and test suites. Reverse engineering is a key to software reengineering [Arnold 1993], because it ensures to recover an abstract representation that can be used for subsequent reengineering of an existing software system.

The goal of reengineering is to reconstitute a software system in a new form that is more evolvable and possibly has more functionality than the original software system. The reengineering process is usually composed of three activities: reverse engineering [Chikofsky and Cross 1990], software restructuring [Arnold 1989] and forward engineering.

- *Reverse engineering* is necessary due to incomplete documentation and relevant references, unavailability of personnel with relevant knowledge, inconsistency between documentation and implementation, outdated technological platforms of a software system, e.g. programming languages, tools and operating systems.
- *Software restructuring* aims to improve certain aspects of a software system and it is “the transformation from one representation form to another at the same relative abstraction level, while preserving the

software system's external behavior, i.e. functionality and semantics" [Yang and Ward 2003].

- *Forward engineering* implements and builds a software system from the restructured model.

This reengineering process is captured in the horseshoe process model for reengineering [Kazman et al. 1998], which consists of three related processes: (i) code and architecture recovery, and conformance evaluation; (ii) architecture transformation; and (iii) architecture-based development in which the new architecture is instantiated.

One approach that assists in software reengineering is refactoring [Fowler 1999], which is a technique for restructuring an existing body of code, altering and improving its internal structure without changing its external behavior. The refactoring process consists of a series of small behavior-preserving transformations. The system is kept fully working after each small refactoring, reducing the chances that a system becomes broken during the restructuring. Refactoring is one way to improve software quality as it helps to improve the design of software, make software easier to understand and help to find bugs [Fowler 1999]. As stated in [Opdyke 1992], while refactorings do not change the behavior of a program, they can support software design and evolution by restructuring a program in a way that allows other changes to be made more easily.

4.8.1 Relation to the Thesis

The software systems that we work with throughout this research are legacy systems that represent valuable software assets. They usually have a long lifetime and most likely have gone through many changes such as technological platform changes and turnover of the original developers. Thus they show signs of many modifications and adaptations. They also have the typical characteristics of legacy systems as described in [Demeyer et al. 2003], e.g. increasing complexity, poor documentation and lack of understanding by the current developers. Therefore, reverse engineering is necessary for understanding the architecture or behavior of a large software system when the source code is the main information. Additionally, as refactoring is one key to increase internal software quality during the whole software lifecycle [Simon et al. 2001], it is one technique that is used in our research when we identify components that need to be refactored and potential architectural improvement proposals to improve the software quality aspects.

4.9 Software Quality Metrics

Various techniques have emerged to qualitatively or quantitatively assess quality impact through specific quality metrics. They differ from each other in terms of principles, concepts and analysis capabilities. For instance, [Kataoka et al. 2002] proposes coupling metrics to measure the maintainability enhancement effect of a program refactoring. [Tahvildari and Kontogiannis 2002] proposes a reengineering transformation framework using soft goal graph to correlate non-functional requirements with design patterns to guide transformation process. The soft goals that are refined from maintainability include coupling, cohesion, modularity, encapsulation, complexity, consistency and reuse. [Tahvildari and Kontogiannis 2003] proposes also another framework which combines using metrics for quality estimation and performing transformation based on soft goal graphs.

To evaluate evolvability, [Ramil and Lehman 2000] proposes metrics based on implementation change logs. [Lehman et al. 1997] proposes computation of metrics using the number of modules in a software system. Another set of metrics is based on software life span and software size [Tamai and Torimitsu 1992]. [Nary and Chung 2003] proposes a framework of process-oriented metrics for software evolvability and traces the metrics back to the evolvability requirements based on the NFR framework [Chung 2000]. An ontological basis which allows for the formal definition of a system and its change at the architectural level is presented in [Rowe and Leaney 1997].

[Simon 1962] describes the link between modularity and evolution, and argues that nearly-decomposable systems facilitate experimentation and problem solving. [LaMantia et al. 2008] examines the design evolution of one open source software product and one company software product platform through the modelling lens of design rule theory and design structure matrices.

4.9.1 Relation to the Thesis

Software evolvability is a multifaceted quality attribute [Rowe et al. 1994], which is refined into a collection of subcharacteristics in our research. Each subcharacteristic is in turn refined into a collection of measuring attributes that we intend to qualitatively and/or quantitatively measure. One particular measuring attribute that we have further explored in our research is modularity. It affects the behavior of a design with respect to most of the evolvability subcharacteristics, as designing software for ease of extension and contraction depends on how well the software structure is organized and modular designs are argued to be more evolvable [Maccormack et al. 2008]. The way that we perform in our case study is similar to the idea in [LaMantia et al. 2008], i.e. through using design rules and design structure matrix. We further enrich the data with experiences and reflections through our dependency analysis of a complex industrial software system.

Chapter 5. Conclusions and Future Work

The goal of the research presented in this thesis is to understand software architecture evolution and to investigate ways to analyze software evolvability to support this evolution. Establishing the evolvability model and systematically assessing the software evolvability at the architecture level are the first steps towards analyzing and quantifying evolvability, a base and check point for evolvability evaluation and improvement. Software architecture evolution is inevitably subject to various change stimuli from technological and business perspectives. Accordingly, comprehensive analysis needs to be performed to obtain knowledge of the potential implications of these change stimuli.

5.1 Contributions

The main contributions of the presented research are summarized as follows:

Software evolvability model. In this thesis, we outline a software evolvability model that provides a basis for analyzing and evaluating software evolvability. This model refines software evolvability into a collection of subcharacteristics that can be measured through a number of measuring attributes. In addition, we further explore one particular measuring attribute, i.e. modularity, which affects the behavior of a design with respect to most of the evolvability subcharacteristics. This is because designing software for ease of extension depends on how well the software structure is organized and modular designs are argued to be more evolvable, i.e. these designs facilitate making future adaptations.

Architecture evolvability analysis method. We introduce a structured method for analyzing evolvability at the architectural level, i.e. the ARchitecture Evolvability Analysis (AREA) method that focuses on improving the capability of being able to on forehand understand and analyze systematically the impact of a change stimulus. The method is studied in an industrial setting.

Comparison analysis framework of CBSE and SOSE. We take component-based and service-oriented software engineering paradigms as an example to analyze a technology-type of change stimulus, i.e. the introduction of SOSE to CBSE. We exemplify the necessity of making analysis and exploration of both the existing and emerging technologies for better understanding of the implications.

Practices in product line migration. We take the adoption of a product line approach as an example to analyze the impacts of a business-type of change stimulus. We focus on managing the migration of legacy systems towards product lines due to the need for differentiation in the marketplace, with short time-to-market as part of the need. Two industrial cases are studied in details. Observations are made with respect to business, organization, development process and technology when adopting a product line approach. The experiences from the case studies are also described to recommend practices that are particularly useful.

Practices in using architecture-level dependency analysis to support software evolution. We explore the links between evolvability, modularity, as well as inter-module dependency, and focus on visualizing static dependencies to identify hotspots in the architecture and implementation, and to provide direction for future improvement. We perform one industrial case study and describe a dependency analysis of a complex industrial power control and protection system, using the inter-module dependency model. Experiences and reflections are made through the analysis process.

5.2 Future Research Directions

A number of potential tracks for further PhD studies and future research are identified as follows:

Further refinement and validation of evolvability model. The initial establishment of the software evolvability model developed in this research has only been motivated and exemplified through one industrial case study. We need to continue working on the evolvability model by conducting more case studies or surveys to confirm and refine the model. A subject that also needs to be investigated is to identify metrics to quantify evolvability subcharacteristics in terms of the identified measuring attributes. In the research presented so far, we have only looked into modularity which is one of the measuring attributes. Further we plan to analyze the correlations among the subcharacteristics with respect to constraints and tradeoffs.

Further validation of evolvability analysis method. The software evolvability analysis method developed in this research has only been exemplified and verified through one industrial case study. Future research includes additional validation of the method using multiple case studies. Another aspect that needs to be considered is to apply the method to address evolvability explicitly in the early design phase of a new development effort, since software architecture that is capable of accommodating change must be specifically designed for change [Isaac and McConaughy 1994].

Further study of the impacts of change stimuli. In this thesis, we have taken the introduction of SOSE to CBSE respective the adoption of product line engineering as examples of technology-type and business-type of change stimuli. Further studies remain to be done to broaden the question at issue and look at other representative change stimuli. An alternative is to enter deeply into the already-selected change stimuli:

- *Further investigation of the impacts of introducing SOSE to CBSE.* In this thesis, we have only partially answered the research question Q3.1 through providing an explicit clarification of the concepts, principles and characteristics of CBSE and SOSE. More work remains to be done to further investigate the impacts of the introduction of SOSE to CBSE.
- *Further study of the adoption of product line engineering*. As product line software engineering has become one of the most established strategies for achieving large-scale software reuse [Estublier and Vega 2005], its impact on software architecture evolution and software evolvability becomes a research area worth further research.

To summarize, future research comprises several tracks that are of different priorities. A top prioritized direction for further research is to further refine and validate the software evolvability model, as it lays a foundation for the rest of the research tracks. This model is a first step towards analyzing and quantifying evolvability, a base and check point for evolvability evaluation and improvement.

References

[Aoyama 2002] Aoyama, M.: 'Metrics and analysis of software architecture evolution with discontinuity', ACM, New York, NY, USA, 2002

[Arnold 1989] Arnold, R.S.: 'Software restructuring', Proceedings of the IEEE, 1989, 77, (4), pp. 607-617

[Arnold 1993] Arnold, R.S.: 'Software reengineering' IEEE Computer Society, Press Los Alamitos, Calif, 1993.

[Avritzer and Weyuker 1999] Avritzer, A. and Weyuker, E.J.: 'Metrics to Assess the Likelihood of Project Success Based on Architecture Reviews', Empirical Software Engineering, 1999, 4, (3), pp. 199-215

[Babar et al. 2004] Babar, M.A., Zhu, L., and Jeffery, R.: 'A framework for classifying and comparing software architecture evaluation methods', Software Engineering Conference, Australian, 2004, pp. 309-318

[Bachmann et al. 2004] Bachmann, F., Goedicke, M., Leite, J., Nord, R., Pohl, K., Ramesh, B., and Vilbig, A.: 'A Meta-model for Representing Variability in Product Family Development', Lecture Notes in Computer Science, 2004, pp. 66-80

[Barais et al. 2004] Barais, O., Cariou, E., Duchien, L., Pessemier, N., and Seinturier, L.: 'TranSAT: A Framework for the Specifcation of Software Architecture Evolution', 2004

[Bass et al. 2003] Bass, L., Clements, P., and Kazman, R.: 'Software Architecture in Practice', Addison-Wesley Professional, 2003.

[Bayer et al. 1999] Bayer, J., Girard, J.F., Wurthner, M., DeBaud, J.M., and Apel, M.: 'Transitioning legacy assets to a product line architecture', ACM, 1999

[Beck 1999] Beck, K.: 'Extreme Programming Explained: Embrace Change', Addison-Wesley, Reading, PA, 1999

[Bengtsson et al. 2004] Bengtsson, P.O., Lassing, N., Bosch, J., and van Vliet, H.: ‘Architecture-level modifiability analysis (ALMA)’, The Journal of Systems & Software, 2004, 69, (1-2), pp. 129-147

[Bennett and Rajlich 2000] Bennett, K. and Rajlich, V.: ‘Software maintenance and evolution: a roadmap’. Proceedings of the Conference on the Future of Software Engineering, Limerick, Ireland, 2000

[Bennett 1996] Bennett, K.: ‘Software evolution: past, present and future’, Information and Software Technology, 1996, 38, (11), pp. 673-680

[Benyon et al. 2005] Benyon, D., Turner, P., and Turner, S.: ‘Designing interactive systems’ Addison-Wesley, New York, 2005.

[Birk et al. 2003] Birk, A., Heller, G., John, I., Schmid, K., von der Massen, T., and Muller, K.: ‘Product line engineering, the state of the practice’, IEEE Software, 2003, 20, (6), pp. 52-60

[Boehm et al. 1978] Boehm, B.W., Brown, J.R., Kaspar, H., Lipow, M., MacLeod, G.J., and Merritt, M.J.: ‘Characteristics of software quality’, North-Holland, 1978.

[Boehm 1988] Boehm, B.W.: ‘A spiral model of software development and enhancement’, Computer, 1988, 21, (5), pp. 61-72

[Bosch 2000] Bosch, J.: ‘Design and use of software architectures: adopting and evolving a product-line approach’, ACM Press/Addison-Wesley Publishing Co., 2000.

[Breivold and Crnkovic 2008] Breivold, H.P. and Crnkovic, I.: ‘Using Software Evolvability Model for Evolvability Analysis’, Mälardalen Real-Time Research Center, Mälardalen University, 2008

[Breivold et al. 2008] Breivold, H.P., Crnkovic, I., and Eriksson, P.J.: ‘Analyzing Software Evolvability’, COMPSAC, 2008

[Brooks 1987] Brooks, F.P.: ‘No Silver Bullet’, IEEE Computer, 1987, 20, (4), pp. 10-19

[Buckley et al. 2004] Buckley, J., Mens, T., Zenger, M., Rashid, A., and Kniesel, G.: ‘Towards a taxonomy of software change’, Journal of Software Maintenance and Evolution: Research and Practice, 2004

[Cervantes and Hall 2004] Cervantes, H. and Hall, R.S.: ‘Autonomous adaptation to dynamic availability using a service-oriented component model’, IEEE Comput. Soc, 2004

[Chapin et al. 2001] Chapin, N., Hale, J.E., Khan, K.M., Ramil, J.F., and Tan, W.G.: 'Types of software evolution and software maintenance', Journal of Software Maintenance and Evolution: Research and Practice, 2001, 13, (1), pp. 3-30

[Chikofsky and Cross 1990] Chikofsky, E.J. and Cross, J.H.: 'Reverse engineering and design recovery: a taxonomy', Software, IEEE, 1990, 7, (1), pp. 13-17

[Christian 2006] Christian, D.R.: 'Continuous evolution through software architecture evaluation: a case study', Journal of Software Maintenance and Evolution: Research and Practice, 2006, 18, pp. 351-383

[Chung 2000] Chung, L.: 'Non-Functional Requirements in Software Engineering', Springer, 2000.

[Clements et al. 2002] Clements, P., Kazman, R., and Klein, M.: 'Evaluating Software Architectures: Methods and Case Studies', Addison-Wesley, 2002.

[Clements and Northrop 2002] Clements, P. and Northrop, L.: 'Software Product Lines: Practices and Patterns. 2002', Addison-Wesley, 2002

[Cockburn 2002] Cockburn, A.: 'Agile Software Development', Addison-Wesley Boston, 2002.

[Crnkovic and Larsson 2002] Crnkovic, I. and Larsson, M.: 'Building Reliable Component-Based Software Systems', Artech House, 2002.

[Demeyer et al. 2003] Demeyer, S., Ducasse, S., and Nierstrasz, O.M.: 'Object-Oriented Reengineering Patterns', Morgan Kaufmann, 2003.

[Dhungana et al. 2006] Dhungana, D., Rabiser, R., Grunbacher, P., Prahofer, H., Federspiel, C., and Lehner, K.: 'Architectural Knowledge in Product Line Engineering: An Industrial Case Study', 32nd EUROMICRO Conference on Software Engineering and Advanced Applications, 2006, pp. 186-197

[Dhungana et al. 2008] Dhungana, D., Neumayer, T., Grünbacher, P., and Rabiser, R.: 'Supporting Evolution in Model-based Product Line Engineering', 12th Int'l Software Product Line Conference, Limerick, Ireland, 2008

[Dobrica and Niemela 2002] Dobrica, L. and Niemela, E.: 'A survey on software architecture analysis methods', IEEE Transactions on Software Engineering, 2002, 28, (7), pp. 638-653

[Dromey 1996] Dromey, R.G.: ‘Cornering the Chimera’, IEEE Software, 1996, 13, (1), pp. 33-43

[Estublier and Vega 2005] Estublier, J. and Vega, G.: ‘Reuse and variability in large software applications’, Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, 2005, pp. 316-325

[Faust and Verhoef 2003] Faust, D. and Verhoef, C.: ‘Software product line migration and deployment’, Software-Practice and Experience, 2003, 33, (10), pp. 933-955

[Fenton and Pfleeger 1997] Fenton, N. and Pfleeger, S.L.: ‘Software metrics: a rigorous and practical approach’, PWS Publishing Co. Boston, MA, USA, 1997.

[Fitzpatrick et al. 2004] Fitzpatrick, R., Smith, P., and O'Shea, B.: ‘Software Quality Challenges’, Proceedings of the Second Workshop on Software Quality at the 26th International Conference on Software Engineering, 2004

[Fowler 1999] Fowler, M.: ‘Refactoring: Improving the Design of Existing Code’, Addison-Wesley Professional, 1999.

[Garlan 2000] Garlan, D.: ‘Software architecture: a roadmap’, ACM Press New York, NY, USA, 2000

[Gay and Airasian 1999] Gay, L.R. and Airasian, P.W.: ‘Educational Research: Competencies for Analysis and Applications’, Prentice Hall, 1999.

[Gilb 1981] Gilb, T.: ‘Evolutionary development [software]’, SIGSOFT Software Engineering Notes, 1981, 6, (2), pp. 17

[Gilb 2002] Gilb, T.: ‘The 10 Most Powerful Principles for Quality in Software and Software Organizations’, Cross-Talk, Nov, 2002

[Grady and Caswell 1987] Grady, R.B. and Caswell, D.L.: ‘Software metrics: establishing a company-wide program’, Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1987.

[Highsmith and Cockburn 2001] Highsmith, J. and Cockburn, A.: ‘Agile Software Development: The Business of Innovation’, 2001

[Highsmith 2000] Highsmith, J.A.: ‘Adaptive software development: a collaborative approach to managing complex systems’, Dorset House Publishing Co., Inc. New York, NY, USA, 2000.

[Holz et al. 2006] Holz, H.J., Applin, A., Haberman, B., Joyce, D., Purchase, H., and Reed, C.: 'Research methods in computing: what are they, and how should we teach them?', Annual Joint Conference Integrating Technology into Computer Science Education, 2006, pp. 96-114

[Isaac and McConaughy 1994] Isaac, D. and McConaughy, G.: 'The Role of Architecture and Evolutionary Development in Accommodating Change', 1994

[ISO9126] ISO9126: 'ISO/IEC 9126-1, International Standard, Software Engineering. Product Quality – Part 1: Quality Model'

[Jansen and Bosch 2004] Jansen, A. and Bosch, J.: 'Evaluation of Tool Support for Architectural Evolution', 2004

[Jansen 2008] Jansen, A.G.J.: 'Architectural Design Decisions', PhD thesis (to appear), 2008

[Jiang and Willey 2005] Jiang, M. and Willey, A.: 'Architecting systems with components and services', Institute of Electrical and Electronics Engineers Computer Society, Piscataway, NJ 08855-1331, United States, 2005

[Kang et al. 1990] Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: 'Feature-Oriented Domain Analysis (FODA) Feasibility Study', the Institute of Software Engineering, 1990.

[Kang et al. 1998] Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M.: 'FORM: A feature-; oriented reuse method with domain-; specific reference architectures', Annals of Software Engineering, 1998, 5, pp. 143-168

[Kataoka et al. 2002] Kataoka, Y., Imai, T., Andou, H., and Fukaya, T.: 'A quantitative evaluation of maintainability enhancement by refactoring', IEEE Comput. Soc, 2002

[Kazman et al. 1994] Kazman, R., Bass, L., Abowd, G., and Webb, M.: 'SAAM: A Method for Analyzing the Properties of Software Architectures', International Conference on Software Engineering, 1994, 16, pp. 81-81

[Kazman et al. 1998] Kazman, R., Woods, S.G., and Carriere, S.J.: 'Requirements for Integrating Software Architecture and Reengineering Models: CORUM II', Working Conference on Reverse Engineering, 1998, pp. 154–163

[Klein et al. 1999] Klein, M., Kazman, R., Bass, L., Carriere, J., Barbacci, M., and Lipson, H.: 'Attribute-Based Architecture Styles', Kluwer, BV Deventer, the Netherlands, 1999.

[Kolb et al. 2005] Kolb, R., Muthig, D., Patzke, T., and Yamauchi, K.: 'A Case Study in Refactoring a Legacy Component for Reuse in a Product Line', Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005, pp. 369-378

[Kotonya et al. 2004] Kotonya, G., Hutchinson, J., and Bloin, B.: 'A Method for Formulating and Architecting Component and Service-Oriented Systems', Stojanovic, Z. et al.(Hrsg.), 2004, pp. 155-181

[Kotonya and Hutchinson 2008] Kotonya, G. and Hutchinson, J.: 'A component-based process for modelling and evolving legacy systems', Software Process Improvement and Practice, 2008, 13, (2), pp. 113-125

[Lago et al. 2008] Lago, P., Avgeriou, P., Capilla, R., and Kruchten, P.: 'Wishes and Boundaries for a Software Architecture Knowledge Community', WICSA, 2008

[LaMantia et al. 2008] LaMantia, M.J., Cai, Y., MacCormack, A., and Rusnak, J.: 'Analyzing the Evolution of Large-Scale Software Systems Using Design Structure Matrices and Design Rule Theory: Two Exploratory Cases', 2008

[LaMantia et al. 2008] LaMantia, M.J., Cai, Y., MacCormack, A.D., and Rusnak, J.: 'Analyzing the evolution of large-scale software systems using design structure matrices and design rule theory: Two exploratory cases', Institute of Electrical and Electronics Engineers Computer Society, Piscataway, NJ 08855-1331, United States, 2008

[Lehman 1980] Lehman, M.M.: 'On understanding laws, evolution, and conservation in the large-program life cycle', Journal of Systems and Software, 1980, 1, (3), pp. 213-221

[Lehman et al. 1997] Lehman, M.M., Ramil, J.F., Wernick, P.D., Perry, D.E., and Turski, W.M.: 'Metrics and laws of software evolution - the nineties view', IEEE Comp Soc, Los Alamitos, CA, USA, 1997

[Lehman et al. 2000] Lehman, M.M., Ramil, J.F., and Kahen, G.: 'Evolution as a noun and evolution as a verb', SOCE 2000 Workshop on Software and Organisation Co-evolution, 2000, pp. 12-13

[Lindvall et al. 2003] Lindvall, M., Tvedt, R.T., and Costa, P.: 'An Empirically-Based Process for Software Architecture Evaluation', Empirical Software Engineering, 2003, 8, (1), pp. 83-108

[Lung et al. 1997] Lung, C.H., Bot, S., Kalaichelvan, K., and Kazman, R.: 'An approach to software architecture analysis for evolution and reusability', IBM Press, 1997

[Maccari and Riva 2002] Maccari, A. and Riva, C.: 'Architectural evolution of legacy product families', Springer-Verlag, 2002

[Maccormack et al. 2008] Maccormack, A., Rusnak, J., and Baldwin, C.Y.: 'the Impact of Component Modularity on Design Evolution: Evidence from the Software Industry', 2008

[Madhavji et al. 2006] Madhavji, N.H., Fernandez-Ramil, J., and Perry, D.: 'Software Evolution and Feedback: Theory and Practice' John Wiley & Sons, 2006.

[Martin 2003] Martin, R.C.: 'Agile Software Development: Principles, Patterns, and Practices', Prentice Hall PTR Upper Saddle River, NJ, USA, 2003.

[Mattsson et al. 2006] Mattsson, M., Grahn, H., and Mårtensson, F.: 'Software Architecture Evaluation Methods for Performance, Maintainability, Testability, and Portability', QoSA, 2006

[McCall et al. 1977] McCall, J.A., Richards, P.K., Walters, G.F., United, S., Electronic Systems, D., Force, A., Rome Air Development, C., and Systems, C.: 'Factors in Software Quality' NTIS, 1977.

[Medvidovic et al. 1998] Medvidovic, N., Taylor, R.N., and Rosenblum, D.S.: 'An Architecture-Based Approach to Software Evolution', 1998

[Mens and Demeyer 2008] Mens, T. and Demeyer, S.: 'Software Evolution' Springer, 2008.

[Nary and Chung 2003] Nary, S. and Chung, L.: 'Process-oriented metrics for software architecture evolvability', IEEE Comput. Soc, 2003

[Nehaniv and Wernick 2007] Nehaniv, C.L. and Wernick, P.: 'Introduction to Software Evolvability', Third International IEEE Workshop on Software Evolvability, 2007

[O'Brien et al. 2007] O'Brien, L., Merson, P., and Bass, L.: 'Quality attributes for service-oriented architectures', Institute of Electrical and

Electronics Engineers Computer Society, Piscataway, NJ 08855-1331, United States, 2007

[O'Reilly 1999] O'Reilly, T.: ‘Lessons from open-source software development’, Communications of the ACM, 1999, 42, (4), pp. 32-37

[Opdyke 1992] Opdyke, W.F.: ‘Refactoring Object-Oriented Frameworks’, University of Illinois, 1992

[Ortega et al. 2003] Ortega, M., Pérez, M., and Rojas, T.: ‘Construction of a Systemic Quality Model for Evaluating a Software Product’, Software Quality Journal, 2003, 11, (3), pp. 219-242

[Palmer and Felsing 2002] Palmer, S. and Felsing, M.: ‘A Practical Guide to Feature Driven Development.’ Prentice Hall, 2002

[Parnas 1994] Parnas, D.L.: ‘Software aging’, Proceedings of 16th International Conference on Software Engineering, 1994, pp. 279-287

[Petriu et al. 2000] Petriu, D., Shousha, C., and Jalnapurkar, A.: ‘Architecture-Based Performance Analysis Applied to a Telecommunication System’, IEEE Transactions on Software Engineering, 2000, pp. 1049-1065

[Pohl et al. 2005] Pohl, K., Böckle, G., and van der Linden, F.: ‘Software Product Line Engineering: Foundations, Principles, and Techniques’ Springer, 2005.

[Ramil and Lehman 2000] Ramil, J.F. and Lehman, M.M.: ‘Metrics of software evolution as effort predictors - a case study’, Institute of Electrical and Electronics Engineers Inc., Piscataway, NJ, USA, 2000

[Reussner et al. 2003] Reussner, R.H., Schmidt, H.W., and Poernomo, I.H.: ‘Reliability prediction for component-based software architectures’, The Journal of Systems & Software, 2003, 66, (3), pp. 241-252

[Rowe et al. 1994] Rowe, D., Leaney, J., and Lowe, D.: ‘Defining systems evolvability-taxonomy of change’, Change, 1994, pp. 541-545

[Rowe and Leaney 1997] Rowe, D. and Leaney, J.: ‘Evaluating evolvability of computer based systems architectures-an ontological approach’, IEEE Computer Society, 1997

[Royce 1987] Royce, W.W.: ‘Managing the development of large software systems: concepts and techniques’, Proceedings of the 9th International Conference on Software Engineering, 1987, pp. 328-338

[Schmid et al. 2005] Schmid, K., John, I., Kolb, R., and Meier, G.: 'Introducing the PuLSE approach to an embedded system population at Testo AG', Association for Computing Machinery, New York, NY 10036-5701, United States, 2005

[Schwaber and Beedle 2001] Schwaber, K. and Beedle, M.: 'Agile Software Development with Scrum', Prentice Hall PTR Upper Saddle River, NJ, USA, 2001.

[Shaw 2002] Shaw, M.: 'What makes good research in software engineering?', International Journal on Software Tools for Technology Transfer (STTT), 2002, 4, (1), pp. 1-7

[SIGCSE] SIGCSE: 'http://www.sigcse.org/', the ACM Special Interest Group on Computer Science Education (SIGCSE)

[Simon et al. 2001] Simon, F., Steinbruckner, F., and Lewerentz, C.: 'Metrics based refactoring', 5th European Conference on Software Maintenance and Reengineering, 2001

[Simon 1962] Simon, H.A.: 'The architecture of complexity', Proceedings of the American Philosophical Society, 1962, 106, (6), pp. 467-482

[Smith et al. 2002] Smith, D., O'Brien, L., and Bergey, J.: 'Using the Options Analysis for Reengineering (OAR) method for mining components for a product line', Springer-Verlag, 2002

[Stapleton 1999] Stapleton, J.: 'DSDM: Dynamic Systems Development Method', Technology of Object-Oriented Languages and Systems, 1999, pp. 406-406

[Stoermer and O'Brien 2001] Stoermer, C. and O'Brien, L.: 'MAP - mining architectures for product line evaluations', IEEE Comput. Soc, 2001

[Stojanovic and Dahanayake 2005] Stojanovic, Z. and Dahanayake, A.: 'Service-oriented Software System Engineering: Challenges and Practices' IGI Global, 2005.

[Tahvildari and Kontogiannis 2002] Tahvildari, L. and Kontogiannis, K.: 'A methodology for developing transformations using the maintainability soft-goal graph', IEEE Comput. Soc, 2002

[Tahvildari and Kontogiannis 2003] Tahvildari, L. and Kontogiannis, K.: 'A metric-based approach to enhance design quality through meta-pattern transformations', IEEE Comput. Soc, 2003

[Tamai and Torimitsu 1992] Tamai, T. and Torimitsu, Y.: ‘Software lifetime and its evolution process over generations’, IEEE Comput. Soc. Press, 1992

[van der Linden et al. 2004] van der Linden, F., Bosch, J., Kamsties, E., Kansala, K., and Obbink, H.: ‘Software product family evaluation’, Springer-Verlag, 2004

[Van Gurp and Bosch 2002] van Gurp, J. and Bosch, J.: ‘Design erosion: problems and causes’, The Journal of Systems & Software, 2002, 61, (2), pp. 105-119

[Wang and Fung 2004] Wang, G. and Fung, C.K.: ‘Architecture paradigms and their influences and impacts on component-based software systems’, Institute of Electrical and Electronics Engineers Computer Society, Piscataway, NJ 08855-1331, United States, 2004

[Wang et al. 1999] Wang, J., He, X., and Deng, Y.: ‘Introducing software architecture specification and analysis in SAM through an example’, Information and Software Technology, 1999, 41, (7), pp. 451-467

[Weiderman et al. 1997] Weiderman, N.H., Bergey, J.K., Smith, D.B., and Tilley, S.R.: ‘Approaches to Legacy System Evolution’, 1997

[Vieira et al. 2000] Vieira, M.E.R., Dias, M.S., and Richardson, D.J.: ‘Analyzing software architectures with Argus-I’, Proceedings of the 22nd international conference on Software engineering, 2000, pp. 758-761

[Williams and Smith 1998] Williams, L.G. and Smith, C.U.: ‘Performance evaluation of software architectures’, Proceedings of the 1st international workshop on Software and performance, 1998, pp. 164-177

[Wohlin and Wesslen 2000] Wohlin, C. and Wesslen, A.: ‘Experimentation in Software Engineering: An Introduction’, Springer, 2000.

[Yang and Ward 2003] Yang, H. and Ward, M.: ‘Successful Evolution of Software Systems’, Artech House, 2003.

[Yau et al. 1978] Yau, S.S., Collofello, J.S., and MacGregor, T.: ‘Ripple effect analysis of software maintenance’, IEEE, 1978

[Yin 2003] Yin, R.K.: ‘Case Study Research: Design and Methods’ Sage Publications Inc, 2003.

[Yu et al. 2008] Yu, L., Ramaswamy, S., and Bush, J.: ‘Symbiosis and Software Evolvability’, IT Professional, 2008, 10, (4), pp. 56-62

[Zelkowitz and Wallace 1997] Zelkowitz, M.V. and Wallace, D.: ‘Experimental validation in software engineering’, Information and Software Technology, 1997, 39, (11), pp. 735-743

Part 2

Paper A

ANALYZING SOFTWARE EVOLVABILITY

Hongyu Pei Breivold, Ivica Crnkovic, Peter J Eriksson

Presented at the 32nd IEEE International Computer Software and Applications Conference (COMPSAC)
Turku, Finland, July 2008

Abstract

Software evolution is characterized by inevitable changes of software and increasing software complexities, which in turn may lead to huge costs unless rigorously taking into account change accommodations. This is in particular true for long-lived systems in which changes go beyond maintainability. For such systems, there is a need to address evolvability explicitly during the entire lifecycle. Nevertheless, there is a lack of a model that can be used for analyzing, evaluating and comparing software systems in terms of evolvability. In this paper, we describe the initial establishment of an evolvability model as a framework for analysis of software evolvability. We motivate and exemplify the model through an industrial case study of a software-intensive automation system.

1. Introduction

Software maintenance and evolution are characterised by their huge cost and cumbersome implementation [1]. The systems' capability to cost-effectively accommodate various changes has become essential. Accordingly, there is a strong need to carry out software evolution efficiently and reliably, and prolong the productive life of a software system. In this paper, we use evolution to refer to the particular evolution stage as described in the staged model by Bennett and Rajlich [1]. We refer to the evolvability definition in [18], since it expresses the dynamic behaviour during a software system's lifecycle and supports the staged model: "*An attribute that bears on the ability of a system to accommodate changes in its requirements throughout the system's lifespan with the least possible cost while maintaining architectural integrity*."

1.1 Motivations

The need to explicitly address software evolvability is becoming recognized [5]. There are examples of different industrial systems that often have a lifetime of 20-30 years. These systems are subject to and may undergo a substantial amount of evolutionary changes, e.g. software technology changes, software systems merge due to organizational changes, demands for distributed development, system migration to product line architecture, etc. The evolution problems we have observed came from different cases. In this paper, we exemplify and analyze in particular one industrial case study that was carried out on a large automation control system at ABB. The controller software consists of more than three million lines of code written in C/C++ and a complex threading model, with support for a variety of different applications and devices. It has grown in size and complexity, as new features and solutions have been added to enhance functionality and to support new hardware, such as devices, I/O boards and production equipment. Such a complex system is difficult to maintain. It is also important and considerably more difficult to evolve. Due to different measures such as organizational and lifecycle process improvements, the system keeps the maintainability, but the evolvability becomes more difficult since the increased complexity in turn leads to decreased flexibility, resulting in problems to add new features. Consequently, it would become costly to adapt to new market demands and penetrate new markets.

Our particular system is delivered as a single monolithic software package, which consists of various software applications developed by distributed development teams. These applications aim for specific tasks in painting, welding, gluing, machine tending and palletizing, etc. In order to keep the integration and delivery process efficient, the initial architectural decision was to keep the deployment artifact monolithic; The complete set of functionality and services is present in every product even though not everything is required in the specific product. As the system grew, it became more difficult to ensure that the modifications of specific application software do not affect the quality of other parts of the software system. As a result, it becomes difficult and time-consuming to modify software artifacts, integrate and test products. To continue exploiting the substantial software investment made and to continuously improve the system for longer productive lifetime, it has become essential to explicitly address evolvability, since the inability to effectively and reliably evolve software systems means loss of business opportunities [1]. We want to emphasize here that the problem raised is not a problem of maintainability. The major problems arise when brand new (very different) features or different development paradigms, shifting business and organizational goals are introduced, so the problems related to the software evolvability – a fundamental element for increasing strategic and economic value of the software [21].

To solve the problems presented above, we need to handle several research issues: (i) which characteristics are necessary for a software system to be evolvable; (ii) how to assess evolvability in a systematic manner; (iii) how to achieve evolvability; and (iv) how to measure evolvability. Accordingly, we outline a software evolvability model in section 2, where necessary subcharacteristics of software evolvability and corresponding measuring attributes are identified. This model is established as a first step towards analyzing and quantifying evolvability, a base and check points for evolvability evaluation and improvement. Further in section 3, we present the structured way of evolvability evaluation that we used in the case study, and a brief analysis of the evolvability subcharacteristics. Section 4 presents related work. Section 5 concludes the paper and outlines the future work.

2. Software evolvability model

Software evolvability is a multifaceted quality attribute [18]. Based on the definition in [18], the software quality challenges and assessment [8], the types of change stimuli and evolution [4], and experiences we gained through industrial case studies, we have discovered that only having a collection of the subcharacteristics of maintainability as defined in the ISO software quality standard [11] is not sufficient for a software system to be evolvable. Therefore, we have (i) complimented and identified subcharacteristics that are of primary importance for an evolvable software system, and (ii) outlined a software evolvability model that provides a basis for analyzing and evaluating software evolvability. The idea with the evolvability model is to further derive the identified subcharacteristics to the extent when we are able to quantify them and/or make appropriate reasoning about the quality of service, as in Figure 1.

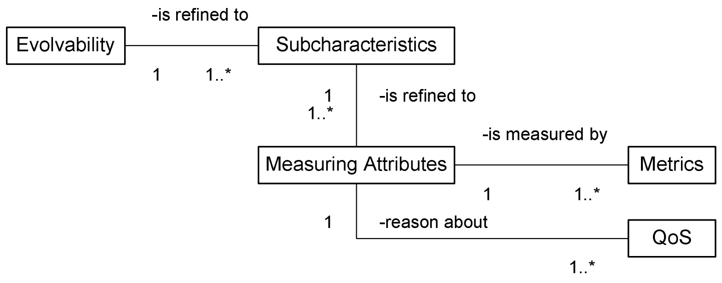


Figure 1. Concept of the evolvability model

The identified subcharacteristics are summarized in Table 1. They are a union of quality characteristics having to do with changes, and are relevant for characterization of evolution of software-intensive systems during their life cycle. With these subcharacteristics in mind, we have a basis on which different systems can be examined and compared in terms of evolvability. Any system that does not explicitly address one or more of these subcharacteristics is missing an element that probably will undermine the system's ability to be evolved.

Table 1. Subcharacteristics of evolvability

Sub-characteristics	**Description**
Analyzability	The capability of the software system to enable the identification of influenced parts due to change stimuli (based on [11]).
Integrity	The non-occurrence of improper alteration of architectural information (based on [12]).
Changeability	The capability of the software system to enable a specified modification to be implemented and avoid unexpected effects (based on [11]).
Extensibility	The capability of the software system to enable the implementation of extensions to expand or enhance the system with new capabilities and features with minimal impact to the existing system (based on [11]).
Portability	The capability of the software system to be transferred from one environment to another [11].
Testability	The capability of the software system to enable modified software to be validated [11].
Domain-specific attributes	The additional quality subcharacteristics that are required by specific domains [8].

These subcharacteristics serve as a catalog of check points for evaluation. Each subcharacteristic is motivated and explained below in conjunction with the case study. Examples of measuring attributes for each subcharacteristic are given.

Analyzability The release frequency of the controller software is twice a year, with around 40 various new requirements that need to be implemented in each release. These requirements may have impact on different attributes of the system, and the possible impact must be analyzed before the implementation of the requirements. This requires that the software system must have the capability to be analyzed and explored in terms of the impact to the software by introducing a change.

Description: Many perspectives are included in this dimension, e.g. identification and decisions on what to modify, analysis and exploration of emerging technologies from maintenance and evolution perspectives.

Measuring attributes include modularity, complexity, and documentation.

Integrity A strategy for communicating architectural principles that we found out from various case studies was to appoint members of the core architecture team as technical leaders in the development projects. However, this strategy although helpful to certain extent, did not completely prevent developers from insufficient understanding and/or misunderstanding of the initial architectural decisions, resulting in violation of architectural conformance. This may lead to evolvability degradation in the long run.

Description: Architectural integrity is related to understanding and coherence to the architectural decisions and adherence to the original architectural styles, patterns or strategies. Taking integrity as one subcharacteristic of evolvability does not mean that the architectural approaches are not allowed to be changed. Proper architectural integrity management is essential for the architecture to allow unanticipated changes in the software without compromising software integrity and to evolve in a controlled way [1].

Measuring attributes include architectural documentation.

Changeability Due to the monolithic characteristic of the controller software, modifications in certain parts of the software package may lead to ripple effects, and requires recompiling, reintegrating and retesting of the whole system. This results in inflexibility of patching and customers have to wait for a new release even in case of corrective maintenance and configuration changes. Therefore, it is required that the software system must have the ease and capability to be changed without negative implications or with controlled implications to the other parts of the software system.

Description: Software architecture that is capable of accommodating change must be specifically designed for change [10].

Measuring attributes include complexity, coupling, change impact, encapsulation, reuse, modularity.

Portability The current controller software supports VxWorks and Microsoft Windows NT. There is a need of openness for choosing among different operating system vendors, e.g. Linux and Windows CE.

Description: Due to the rapid technical development on hardware and software technologies, portability is one of the key enablers that can provide possibility to choose between different hardware and operating system vendors as well as various versions of frameworks.

Measuring attributes include mechanisms facilitating adaptation to different environments.

Extensibility The current controller software supports around 20 different applications that are developed by several distributed development centers around the world. To adapt to the increased customer focus on specific applications and to enable establishment of new market segments, the controller, like any other software systems, must constantly raise the service level through supporting more functionality and providing more features [3].

Description: One might argue that extensibility is a subset of changeability. Due to the fact that about 55% of all change requests are new or changed requirements [15], we define extensibility explicitly as one subcharacteristic of evolvability. It is a system design principle where the implementation takes future growth into consideration.

Measuring attributes include modularity, coupling, encapsulation, change impact.

Testability The controller software exposed huge number of public interfaces which resulted in tremendous time merely on interface tests. One task was therefore to reduce the public interfaces to around 10%. Besides, due to the monolithic characteristic, error corrections in one part of the software requires retesting of the whole system. One issue was therefore to investigate the feasibility of testing only modified parts.

Description: According to statistics [7], software testing spends as much as 50% of development costs and comprises up to 50% of development time. Hence, testability is a key feature permitting high quality to be combined with reduced time-to-market.

Measuring attributes include complexity, modularity.

Domain- specific attributes The controller software has critical real-time calculation demands. It is also required to reduce base software code size and runtime footprint.

Description: Different domains may require additional quality characteristics that are specific for a software system to be evolvable.

Measuring attributes depend on the specific domains.

3. Case Study

We conducted the following structured evaluation steps shown in Figure 2. The involved stakeholders expressed that they were pleased with this systematic approach, as it made architecture requirements and corresponding design decisions more explicit, better founded and documented.

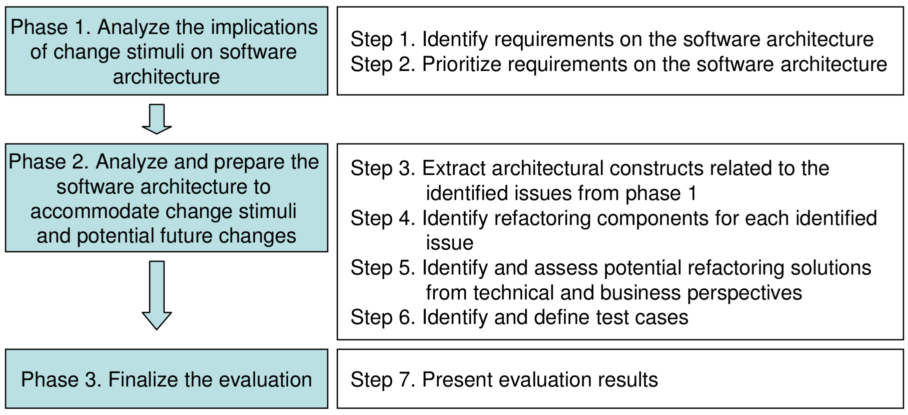

Phase 1. Analyze the implications of change stimuli on software architecture
Step 1. Identify requirements on the software architecture
Step 2. Prioritize requirements on the software architecture
Phase 2. Analyze and prepare the software architecture to accommodate change stimuli and potential future changes
Step 3. Extract architectural constructs related to the identified issues from phase 1
Step 4. Identify refactoring components for each identified issue
Step 5. Identify and assess potential refactoring solutions from technical and business perspectives
Step 6. Identify and define test cases
Phase 3. Finalize the evaluation
Step 7. Present evaluation results

Figure 2. Evaluation steps

The evaluation results included (i) the identified and prioritized requirements on the software architecture; (ii) identified components/modules that need to be refactored for enhancement or adaptation; (iii) refactoring investigation documentation which describes the current situation and solutions to each identified candidate that need to be refactored, including estimated workload; and (iv) test scenarios.

3.1 Analysis of evolvability subcharacteristics

Analyzability was addressed through refining activities for each identified requirement. **Integrity** was addressed through extracting rationale for each design decision; and providing training, guidelines and code examples for software developers and using tactics that enable the achievement of a certain quality characteristic. **Changeability** was addressed through restructuring the original function-oriented architecture to product-line architecture. **Extensibility** was addressed through the definition of a Base Software SDK (Software Development Kit), consisting of well-documented API (Application Programming Interface), wizards and tools for developing application-specific extensions. **Portability** was handled through the portability layer which encapsulates infrastructure technology choices and provides interfaces for application software in the controller. **Testability** was addressed through defining test scenarios and applications to support platform testing. **Domain-specific attribute** was planned with respect to functionality partition of the controller software.

4. Related work

To evaluate evolvability, Ramil and Lehman proposed metrics based on implementation change logs [16] and computation of metrics using the number of modules in a software system [13]. Another set of metrics is based on software life span and software size [20]. In [19], a framework of process-oriented metrics for software evolvability was proposed to intuitively develop architectural evolvability metrics and to trace the metrics back to the evolvability requirements based on the NFR framework. The best known quality models for evaluating quality include McCall [14], Boehm [2], FURPS [9], ISO 9126 [11] and Dromey [6]. However, the term evolvability is not explicitly addressed in any of the quality models. An ontological basis which allows for the formal definition of a system and its change at the architectural level is presented in [17]. [18] proposed a taxonomy to address change as factors and classify evolvability into several aspects, e.g. generality, adaptability, scalability and extensibility. However, it does not cover all the types of software evolution, e.g. concerns of product line development.

5. Conclusions and future work

This paper proposes and demonstrates an evolvability model and an evaluation approach, which were applied into complex industrial context to assist software evolvability analysis. By establishing the evolvability model, we hope to have improved the capability in being able to on forehand understand and analyze systematically the impact of a change stimulus. This, in turn, helps us to prolong the evolution stage.

We intend to continue working on the evolvability model by conducting more case studies to confirm and refine the model. Further we plan to analyze the correlations among the subcharacteristics with respect to constraints and tradeoffs.

References

[1] Bennett, K. and Rajlich, V., “Software Maintenance and Evolution: a Roadmap”, The Future of Software Engineering, Anthony Finkelstein (Ed.), ACM Press 2000.

[2] Boehm, B. W. et al., Characteristics of Software Quality, Amsterdam, North-Holland, 1978.

[3] Bosch, J., Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach, Addison-Wesley, 2000.

[4] Chapin, N. et al., “Types of Software Evolution and Software Maintenance”, Journal of Software Maintenance and Evolution: Research and Practice, 2001.

[5] Ciraci, S. and Broek, P., “Evolvability as a Quality Attribute of Software Architectures”, the International ERCIM Workshop on Software Evolution, 2006.

[6] Dromey, G., “Cornering the Chimera”, IEEE Software (January): 33-43, 1996.

[7] Eickelmann, N. S. and Richardson, D. J., “What Makes One Software Architecture More Testable Than Another”, SIGSOFT Workshop, 1996.

[8] Fitzpatrick, R. et al., “Software Quality Challenges”, 26th International Conference on Software Engineering, 2004.

[9] Grady, R. and Caswell, D., Software Metrics: Establishing a Company-Wide Program, Englewood Cliffs, NJ, PrenticeHall, 1987.

[10] Isaac, D. and McConaughy, G., “The Role of Architecture and Evolutionary Development in Accommodating Change”, Proceedings of NCOSE’94, 1994.

[11] ISO/IEC 9126-1, International Standard, Software Engineering, Product Quality – Part 1: Quality Model, 2001.

[12] Laprie, Dependable Computing and Fault-Tolerant Systems. Vol. 5, Dependability: Basic Concepts and Terminology. Laprie, J.C. (ed.). New York: Springer, 1992.

[13] Lehman, M. M. and Ramil, J. F. et al., “Metrics and Laws of Software Evolution – The Nineties View”, IEEE Computer Press, pp 20-32, 1997.

[14] McCall, J. A., Richards, P. K. and Walters, G. F., Factors in Software Quality, National Technical Information Service, 1977.

[15] Pigoski, T. M., Practical Software Maintenance, Wiley Computer Publishing, 1996.

[16] Ramil, J. F. and Lehman, M. M., “Metrics of Software Evolution as Effort Predictors – A Case Study”, ICSM, 2000.

[17] Rowe, D. and Leaney, J., “Evaluating Evolvability of Computer Based Systems Architectures – an Ontological Approach”, Proceedings of International Conference and Workshop on Engineering of Computer-Based Systems, 1997.

[18] Rowe, D. and Leaney, J., “Defining Systems Evolvability – a Taxonomy of Change”, Proceedings of the IEEE Conference on Computer Based Systems, 1998.

[19] Subramanian, N. and Chung, L., “Process-Oriented Metrics for Software Architecture Evolvability”, 6th IWPSE, 2002.

[20] Tamai, T. and Torimitsu, Y., “Software Lifetime and its Evolution Process over Generations”, ICSM, 1992.

[21] Weiderman, N. H. et al., “Approaches to Legacy Systems Evolution”, Technical Report CMU/SEI-97-TR-014, 1997.

Paper B

ANALYZING SOFTWARE EVOLVABILITY OF AN INDUSTRIAL AUTOMATION CONTROL SYSTEM: A CASE STUDY

Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land, Magnus Larsson

Presented at the 3rd International Conference on Software Engineering Advances (ICSEA)
Sliema, Malta, October 2008

Abstract

Evolution of software systems is characterized by inevitable changes of software and increasing software complexity, which in turn may lead to huge maintenance and development costs. For long-lived systems, there is a need to address evolvability (i.e. a system's ability to easily accommodate changes) explicitly in the requirements and early design phases, and maintain it during the entire lifecycle. This paper describes our work in analyzing and improving the evolvability of an industrial automation control system, and presents 1) evolvability subcharacteristics based on the problems in the case and available literature; 2) a structured method for analyzing evolvability at the architectural level - the ARchitecture Evolvability Analysis (AREA) method. This paper includes also the main analysis results and our observations during the evolvability analysis process in the case study. The evolvability subcharacteristics and the method should be generally applicable, and they are being validated within another domain at the time of writing.

1. Introduction

Studies indicate that more than 50% of the total life cycle cost is spent after the initial development [18]. Therefore, it becomes essential to cost-effectively carry out software evolution. In order to prolong the productive life of a software system, the need to explicitly address software evolvability is becoming recognized [6]. There are examples of industrial systems with a lifetime of 20-30 years. These systems are subject to and may undergo a substantial amount of evolutionary changes, e.g. shifting business and organizational goals, software technology changes, software systems merge due to organizational changes [16], demands for distributed development, system migration to product line architecture, etc. The evolution problems we have observed came from various cases in industrial context, where evolvability was identified as a very important quality attribute that must be maintained. In order to preserve and improve evolvability, we need to (i) analyze the system with respect to evolvability; and (ii) perform architectural transformation. It is generally acknowledged that the software's architecture holds a key to the possibility to implement changes in an efficient manner [1]. Therefore, in this paper, we analyze evolvability at the architecture level and identify the evolvability subcharacteristics of interest in an industrial case study, where a large automation control system at ABB was evolved from a monolithic architecture towards a product line. We present our experiences of the development of the product line architecture in the form of a general method, which we have constructed from data in the manner of *grounded theory research* [25]. In addition, the risk of bias has been further decreased through the involvement of other researchers in the analysis of the experiences.

The remainder of this paper is structured as follows. Section 2 describes the context of the case study. Section 3 presents our architecture evolvability analysis method - AREA. Section 4 presents the case study, in which the method was applied to analyze, evaluate and improve the software architecture of the automation controller software system. Section 5 discusses the experiences we gained through the case study. Section 6 reviews related work. Section 7 concludes the paper.

2. Context of the case

This section presents the case to motivate evolvability analysis and describe seven evolvability subcharacteristics from the case perspective.

2.1 Motivating Evolvability Analysis

The case study was based on a large automation control system at ABB and focused on the latest generation of the controller. The controller software consists of more than three million lines of code written in C/C++ and uses a complex threading model, with support for a variety of different applications and devices. It has grown in size and complexity, as new features and solutions have been added to enhance functionality and to support new hardware, such as devices, I/O boards and production equipment. Such a complex system is difficult to maintain. It is also important and considerably more difficult to evolve. Due to different measures such as organizational and lifecycle process improvements, the system keeps the maintainability, but the evolvability becomes more difficult since the increased complexity in turn leads to decreased flexibility, resulting in problems to add new features. Consequently, it becomes costly to adapt to new market demands and penetrate new markets.

Our particular system is delivered as a single monolithic software package, which consists of various software applications developed by distributed development teams. These applications aim for specific tasks in painting, welding, gluing, machine tending and palletizing, etc. To keep the integration and delivery process efficient, the initial architectural decision was to keep the deployment artifact monolithic. The complete set of functionality and services is present in every product even though not everything is required in the specific product. As the system grew, it became more difficult to ensure that the modifications of specific application software do not affect the quality of other parts of the software system. As a result, it became difficult and time-consuming to modify software artifacts, integrate and test products. To continue exploiting the substantial software investment made and to continuously improve the system for longer productive lifetime, it has become essential to explicitly address evolvability, since software evolvability is a fundamental element for increasing strategic and economic value of the software [28]. The inability to effectively and reliably evolve software systems means loss of business opportunities [2].

2.2 Evolvability Subcharacteristics from Case Perspective

In our previous work [21], we have identified subcharacteristics that are of primary importance for an evolvable software system. Definitions and detailed explanations of evolvability subcharacteristics are provided in [21]. The derivation of evolvability subcharacteristics are based on survey and analysis of literatures (see related work section), problems we have observed and experiences from several earlier case studies. We do not exclude the possibilities that other domains or cases might have slightly extended set of subcharacteristics. Each subcharacteristic is explained below in conjunction with the case.

Analyzability The release frequency of the controller software is twice a year, with around 40 various new major requirements that need to be implemented in each release. These requirements have impact on different attributes of the system, and the possible impact must be analyzed before the implementation of the requirements. This requires that the software system must have the capability to be analyzed and explored in terms of the impact to the software by introducing a change.

Architectural Integrity A strategy for communicating architectural decisions that we found out from various case studies was to appoint members of the core architecture team as technical leaders in the development projects. However, this strategy although helpful to certain extent, did not completely prevent developers from insufficient understanding and/or misunderstanding of the initial architectural decisions, resulting in unconscious violation of architectural conformance. This may lead to evolvability degradation in the long run. Therefore, it is important to record rationale for each design decision, strategy and architectural solution.

Changeability Due to the monolithic characteristic of the controller software, modifications in certain parts of the software package lead to some ripple effects, and requires recompiling, reintegrating and retesting of the whole system. This results in inflexibility of patching and customers have to wait for a new release even in case of corrective maintenance and configuration changes. Therefore, it is strongly required that the software system must have the ease and capability to be changed without negative implications or with controlled implications to the other parts of the software system.

Portability The current controller software supports VxWorks and Microsoft Windows NT. There is a need of openness for choosing among different operating system (OS) vendors, e.g. Linux and Windows CE, and possibly new OS in the future.

Extensibility The current controller software supports around 20 different applications that are developed by several distributed development centers around the world. To adapt to the increased customer focus on specific applications and to enable establishment of new market segments, the controller, like any other software systems, must constantly raise the service level through supporting more functionality and providing more features [4], while keeping some important extra-functional properties, such as performance, or reliability.

Testability The controller software exposed huge number of public interfaces which resulted in tremendous time merely on interface tests. One task was therefore to reduce the public interfaces to around 10% of the original public interfaces. Besides, due to the monolithic characteristic, error corrections in one part of the software requires retesting of the whole system. One issue was therefore to investigate the feasibility of testing only modified parts.

Domain- specific attributes The controller software has critical real-time calculation demands. It is also expected to reduce the base software code size and runtime footprint.

3. Overview of the ARchitecture Evolvability Analysis (AREA) method

The steps that we performed in the case are divided into three main phases as shown in Figure 1.

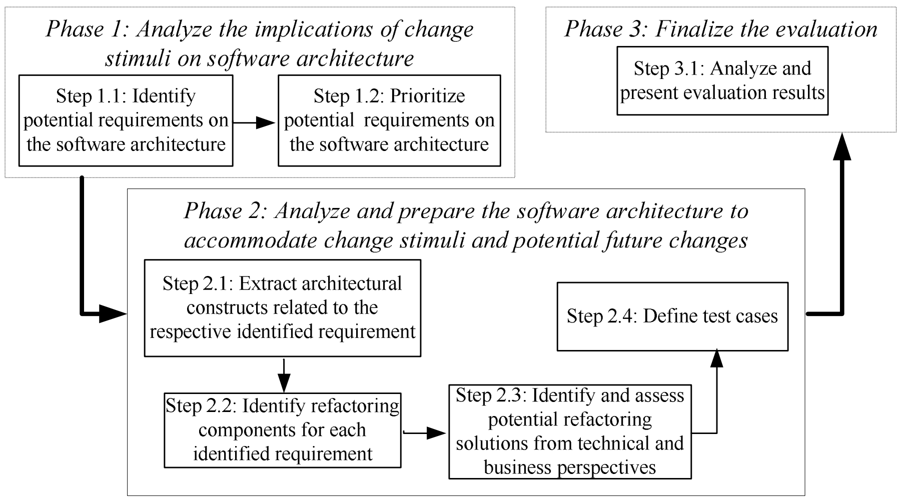

Phase 1: Analyze the implications of change stimuli on software architecture
Step 1.1: Identify potential requirements on the software architecture
Step 1.2: Prioritize potential requirements on the software architecture
Phase 3: Finalize the evaluation
Step 3.1: Analyze and present evaluation results
Phase 2: Analyze and prepare the software architecture to accommodate change stimuli and potential future changes
Step 2.1: Extract architectural constructs related to the respective identified requirement
Step 2.4: Define test cases
Step 2.2: Identify refactoring components for each identified requirement
Step 2.3: Identify and assess potential refactoring solutions from technical and business perspectives

Figure 1. Steps of ARchitecture Evolvability Analysis (AREA) method

***Phase 1**: Analyze the implications of change stimuli on software architecture.*

This phase analyzes the architecture for evolution and understands the impact of change stimuli on the current architecture. Software evolvability concerns both business and technical issues [29], since the stimuli of changes come from both perspectives, e.g. environment, organization, process, technology and stakeholders' needs. These change stimuli have impact on the software system in terms of software structures and/or functionality.

Step 1.1: *Identify potential requirements in the software architecture.*

Any change stimulus results in a collection of potential requirements that the software architecture needs to adapt to. The aim of this step is to extract these requirements that are essential for software architecture enhancement so as to cost-effectively accommodate to change stimuli. Architecture workshops can be conducted, where the stakeholders discuss and identify the potential architecture requirements. Each requirement is concretized with a collection of identified refined activities. Afterwards, each identified requirement must be checked against the evolvability subcharacteristics so as to ensure the consistency and completeness.

Step 1.2: *Prioritize potential requirements in the software architecture.*

In order to establish a basis for common understanding of the architecture requirements among stakeholders within the organization, all the potential requirements identified from the first step need to be prioritized. We do not propose any general criteria for requirement prioritization that apply to all the software systems evolution, since the criteria might be different from case to case depending on factors such as development and organizational constraints, the probability of potential requirements becoming mandatory requirements that the architecture must adapt to, etc.

Phase 2*: Analyze and prepare the software architecture to accommodate change stimuli and potential future changes.*

This phase focuses on the identification and improvement of the components that need to be refactored.

Step 2.1: *Extract architectural constructs related to the respective identified requirement.*

We mainly focus on architectural constructs that are related to each identified requirement. In order for the architecture to allow changes in the software without compromising software integrity and to evolve in a controlled way, documentation of architectural decisions and their rationale play a key role.

Step 2.2: *Identify refactoring components for each identified requirement.*

In this step, we identify the components that need refactoring in order to fulfill the prioritized requirements.

Step 2.3: *Identify and assess potential refactoring solutions from technical and business perspectives.*

Refactoring solutions are identified and design decisions are taken in order to fulfill the requirements derived from the first phase. The change propagation of the effect of refactoring need to be considered and provided as an input to the business assessment, estimating the cost and effort on applying refactorings. In some cases, the refactoring of a certain component is straightforward if we know how to refactor with only local impact. When the implementation is uncertain and might affect several subsystems or modules, prototypes need to be made to investigate the feasibility of potential solutions as well as the estimation of implementation workload. As part of this step, an assessment regarding the compatibility of the refactoring solutions and rationale with earlier made design decisions is made to ensure architectural integrity.

Step 2.4: *Define test cases.*

New test cases that cover the affected component, modules or subsystems need to be identified.

Phase 3*: Finalize the evaluation.*

In this phase, the previous results are incorporated, analyzed and structured into a collection of documents.

Step 3.1*: Analyze and present evaluation results.*

The evaluation results include (i) the identified and prioritized requirements on the software architecture; (ii) the identified components/modules that need to be refactored for enhancement or adaptation; (iii) refactoring investigation documentation which describes the current situation, rationale and solutions to each identified candidate that need to be refactored, including estimated workload; (iv) test scenarios; and (v) impact analysis on evolvability.

4. Applying the AREA method

The main focus of the analysis in our case was to assess how well the architecture would support potential forthcoming requirements and understand their impact. Through the analysis process, we identified potential flaws and defined an evolution path of the software system. The identification and analysis of the architectural requirements was performed by the architecture core team which consists of 6-7 persons. It was a continuous maturation process from the first vision to concrete activities that took approximately one calendar year including analysis, identification of architecture evolution path and partial refactoring. 2-3 persons from the architecture core team identified the refactoring solution proposals for the components in the *Basic Services* subsystem. These proposals were discussed with the main technical responsible persons and architects, documented as evolution path for the architecture and transferred further to the implementation teams.

4.1 Phase 1 - Step 1.1: Identify potential requirements on the software architecture

The change stimuli to the controller software came from the following emerging critical issues related to software evolution: (a) time-to-market requirements, such as building new products for dedicated market within short time; (b) improvement of software system evolvability; and (c) increased ease and flexibility of distributed development of products in combination with the diversity of application variants. We list below the main potential architecture requirements that were identified from the change stimuli. The refined activities for each requirement are presented as well.

R1. Improved modularization of architecture.

a) Enable the separation of layers within the controller software: (i) a kernel which comprises of components that must be included by all application variants; (ii) common extensions which are available to and can be selected by all application variants; and (iii) application extensions which are only available to specific application variants.
b) Investigate dependencies between the existing extensions.

R2. Reduced architecture complexity.

a) Define interfaces and reduce public interface calls.
b) Add support for task isolation and task management.

R3. Enable distributed development of extensions with minimum dependency.

a) Build the application-specific extensions on top of the base software (kernel and common extensions) without the need of modification to the internal base source code.
b) Package the base software into SDK (Software Development Kit), which provides necessary interfaces, tools and documentation to support distributed application development and separate release cycles of the SDK and application-specific extensions.

R4. Improved portability.

Investigate portability across target operating system platforms and across hardware platforms.

R5. Impact on product development process.

a) Investigate the implications of software restructuring on product integration and testing.

R6. Minimized software code size and runtime footprint.

a) Investigate enabling mechanisms, e.g. properly partitioning functionality.

The above architecture requirements should be checked against the evolvability subcharacteristics to justify whether the realization of each requirement would lead to an improvement of the subcharacteristics (or possibly a decrease, which would then require a tradeoff decision), as summarized in Table 1. Besides, the choice of component refactoring and implementation solution proposals for fulfilling each requirement might cause tradeoffs against some other subcharacteristics, as detailed in section 4.7.

Table 1. Mapping between evolvability subcharacteristics and architecture requirements

Subcharacteristics	**Requirements**
Analyzability	R1. Improved modularization of architecture. R2. Reduced architecture complexity.
Architectural Integrity	not related to any particular architectural requirement, but rather to whether the architectural choices and rationale for handling these requirements are documented
Changeability	R1. Improved modularization of architecture. R2. Reduced architecture complexity.
Extensibility	R3. Enable distributed development of extensions with minimum dependency.
Portability	R4. Improved portability.
Testability	R5. Impact on product development process.
Domain-specific attributes	R6. Minimized software code size and runtime footprint.

4.2 Phase 1 - Step 1.2: Prioritize potential requirements on the software architecture

Due to the monolithic characteristics of the architecture, the individual products are burdened with functionalities and components that are not necessary for the specific individual products. Accordingly, the main idea was to apply the product line approach, transform the existing system into reusable components that can form the core of the product-line infrastructure, and separate application-specific extensions from the base software. With the consideration of not disrupting the ongoing development projects, the criteria for requirement prioritization were: (i) enable building of existing types of extensions after refactoring and architecture restructuring; (ii) enable new extensions and simplify interfaces that are difficult to understand and may have negative effects on implementing new extensions. Based on these criteria, R1, R2 and R3 were prioritized potential architectural requirements.

4.3 Phase 2 - Step 2.1: Extract architectural constructs related to the respective identified requirement

Over years of development, a lot of functionality has been added to the system to support new requirements. It becomes easy to unconsciously violate the original good design decisions. To prevent this, it is important to extract design decisions and rationale through documentation of architectural constructs. In this way, potential architectural flaws can be discovered. For instance, in the case study, some implementation violations were discovered, such as improper use of conditional compilation in case of environment changes, direct access to OS native APIs, etc. Additional efforts have been put to provide training, guidelines/rules and code examples for software developers in writing code and using tactics that enable the achievement of a certain quality characteristic. We exemplify with R3 and extract architectural constructs in form of the original coarse-grained architecture as depicted in Figure 2.

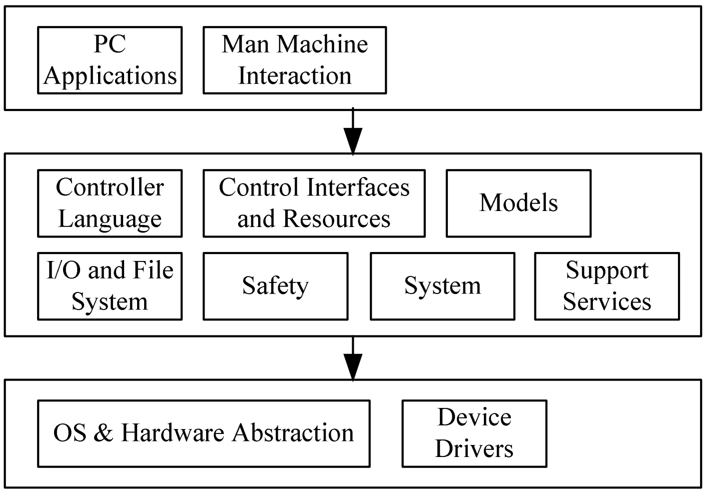


Figure 2. A conceptual view of the original software architecture

The lower layer provides an interface to the upper layer and allows the source code of the upper layer to be used on different hardware platforms and operating systems. The main problem with this software architecture was the existence of tight coupling among some components that reside in different layers. This led to additional work required at a lower level to modify some existing functionality and add support for new functionality in various applications. For instance, the system is required to perform certain tasks during start-up and shutdown in the controller. Some routines for handling such tasks had to be hard-coded, i.e. the application developers had to edit in the source code of e.g. *Support Services* subsystem in the lower layer, which is developed by another group of developers. Accordingly, source code updates had to be done not only on the application level, but through several layers, several subsystems and components. Recompilation of the whole code base was required. This required that application developers need to have a thorough knowledge of the complete source code. It also constituted a bottleneck in the effort to enable distributed application development.

4.4 Phase 2 - Step 2.2: Identify refactoring components for each identified requirement

To cope with the architectural problems identified in the previous step, the strategy of separate concerns need to be applied to isolate the effect of changes to parts of the system [11], i.e. separate the global functions from the hardware, and separate application-specific functions from generic and basic functions as illustrated in Figure 3.

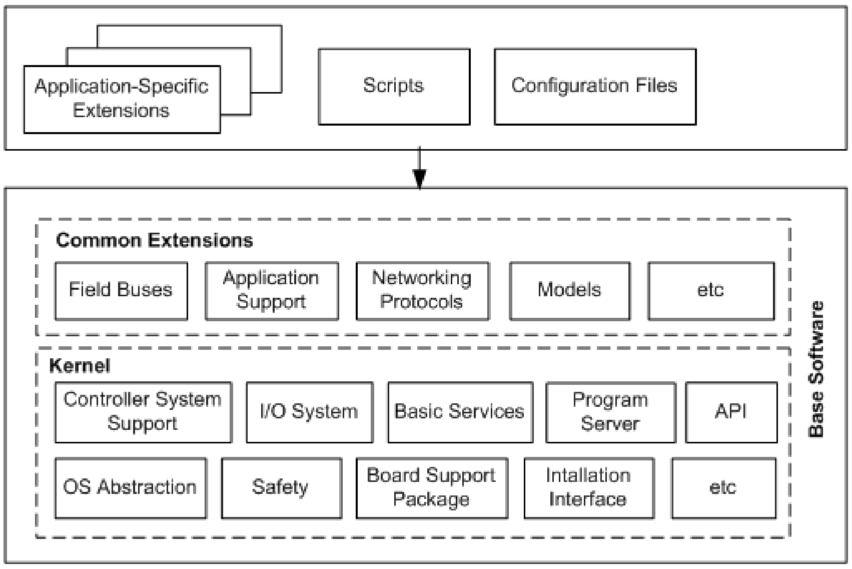


Figure 3. A revised conceptual view of the software architecture

Accordingly, some components need to be adapted and reorganized to enable the architecture restructuring, e.g. some components within the low-level *Basic Services* subsystem for resource allocations, including semaphore ID management component, memory allocation management component to separate functionality from resource management and to achieve the build- and development-independency between the kernel and extensions.

4.5 Phase 2 - Step 2.3: Identify and assess potential refactoring solutions from technical and business perspectives

Due to space limitations and company confidentiality, we exemplify with one component example (inter-process communication component) that needed to be refactored to represent and illustrate for the many various discussions and solutions that occurred during the analysis. We discuss in terms of the following views: (i) problem description: the problem and disadvantages of the original design of the component; (ii) requirements: the new requirements that the component needs to fulfill; (iii) improvement solution: the architectural solution to design problems; (iv) rationale and architectural consequences: the rationale of the solution proposal and architectural implications of the deployment of the component on quality attributes; and (v) estimated workload: the estimated workload for implementation and verification.

4.5.1. Inter-Process Communication

This component belongs to Basic Services subsystem and it includes mechanisms that allow communication between processes, such as remote procedure calls, message passing and shared data.

Problem Description. All the slot names and slot IDs that are used by the kernel and extensions are defined in a C header file in the system. The developers have to edit this file to register their slot name and slot ID, and recompile. Afterwards, both the slot name and slot ID have to be specified in the startup command file for thread creation. There is no dynamic allocation of connection slot.

Requirements. The refactoring of this component is related to R3. It should be possible to define and use IPC slots in common extensions and application extensions without the need to edit the source code of the base software and recompile. The mechanism for using IPC from extensions must be available also in the kernel, to facilitate move of components from kernel to extensions in the future.

Improvement Solution. The slot ID for extension clients should not be booked in the header file. Extensions should not hook a static slot ID in the startup command file. The command attribute dynamic slot ID should be used instead. The IPC connection for extension clients will be established dynamically through the `ipc_connect` function as shown in Figure 4. It will return a connection slot ID when no predefined slot ID is given. An internal error will be logged at startup if a duplicate slot name is used.

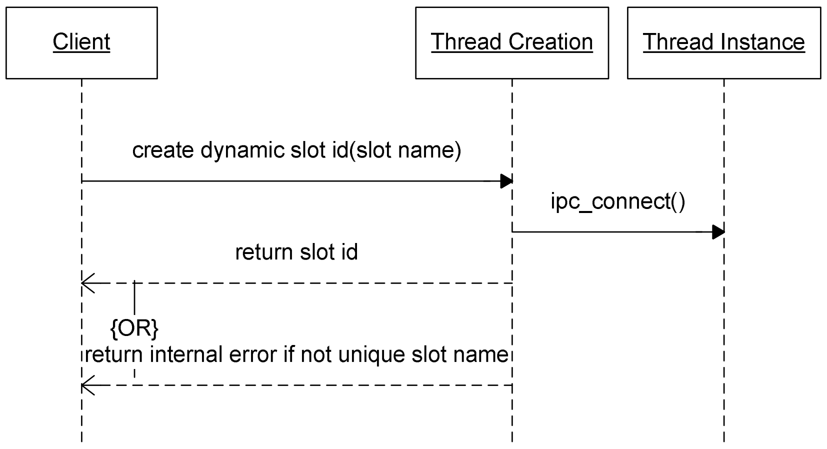

Client
Thread Creation
Thread Instance
create dynamic slot id(slot name)
ipc_connect()
return slot id
{OR}
return internal error if not unique slot name

Figure 4. The inter-communication component after refactoring

Rationale and Architectural Consequences. The revised IPC component provides efficient resource booking for inter-process communication and enables encapsulation of IPC facilities. Accordingly, distributed development of extensions utilizing IPC functionality is facilitated. The use of dynamic inter-process communication connections addressed resource limitations for IPC connection. In this way, limited IPC resources are used only when the processes are communicating. However, the use of IPC mechanisms requires resources, which are limited on a real-time operating system. Therefore, the overheads due to resource description processing may be the offset against efficiency [22], since the overall real-time performance may be degraded if the cost of creating and destroying IPC connections is too high.

Estimated Workload. It was estimated around 2 man weeks which includes the IPC component refactoring and moving IPC client from kernel to extension.

4.6 Phase 2 - Step 2.4: Define test cases

The corresponding test cases were derived based on the selected improvement solution proposal to each component that needed refactoring. For instance, the architectural test cases for the IPC component are given by the ThreadCreation class creating dynamic slot ID, as shown in Figure 5.

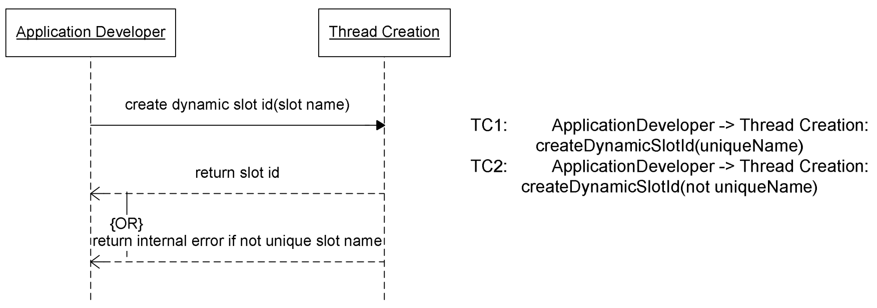

Application Developer
Thread Creation
create dynamic slot id(slot name)
return slot id
{OR}
return internal error if not unique slot name
TC1: ApplicationDeveloper -> Thread Creation: createDynamicSlotId(uniqueName)
TC2: ApplicationDeveloper -> Thread Creation: createDynamicSlotId(not uniqueName)

Figure 5. Test cases for IPC management component

4.7 Phase 3 - Step 3.1: Present evaluation results

In this step, the implications of the potential improvement strategies and evolution path of the software architecture are analyzed with respect to the evolvability subcharacteristics as illustrated in Table 2.

Table 2. Impacts of the IPC component on evolvability subcharacteristics (+ positive impact, - negative impact)

	Consequences of changing IPC component
Analyzability	– due to less possibility of static analysis since definitions are defined dynamically
Architectural Integrity	+ due to documentation of specific requirements, architectural solutions and consequences
Changeability	+ due to the dynamism which makes it easier to introduce and deploy new slots
Portability	+ due to improved abstraction of Application Programming Interfaces (APIs) for IPC
Extensibility	+ due to encapsulation of IPC facilities and dynamic deployment
Testability	No impact
Domain-specific attributes	+ resource limitation issue is handled through dynamic IPC connection – due to introduced dynamism, the system performance could be slightly reduced

5. Reflections

This section summarizes our observations and experiences of applying AREA.

5.1 Experiences

By applying AREA method, we have improved the capability in being able to on forehand understand and analyze systematically the impact of a change stimulus. This, in turn, helps us to prolong the evolution stage [2]. Besides, we list below two observations that concern visible improvements in the organization. They were perceived and informally reported by the stakeholders themselves.

Documentation of architecture is improved, including the architecture's evolution path. Architecture transformation and suggestions for refactoring solutions were part of the analysis process. This was performed by the architecture core team. As a result of the analysis and refactoring activities, the documentation of design and implementation solution proposals has been improved. The final refactoring analysis investigation report was distributed for inspection and was approved after a few iterations. This document served as an input and blueprint to the implementation teams. In this way, the architecture core team and implementation teams shared the same view on the evolution path of the software architecture.

High-level business goals lead to architectural requirements. In the case study, the potential requirements on the architecture were derived from the high-level business goals through the first phase, where the potential requirements on the architecture were identified based on the change stimuli. Such derivation provides an understanding on how the intended software system and its evolving artifacts reflect and contribute to the strategic goals. Together with the documentation of architecture evolution path, it would enrich architectural models and facilitate the traceability of software architecture evolution back to the various business constraints and assumptions [15].

5.2 Suggestions

Due to continuously changing requirements and evolutions of new technologies, the software architecture needs to be evolvable to cost-effectively accommodate changes. Thus, we suggest routine evolvability analysis that should be applied as an integral part during the whole software lifecycle.

Another remark is that the process of making the impact analysis of component refactoring in terms of estimated workload was not an easy task. One principle that was applied during the component refactoring process was to preserve the external behavior of the system despite the number of changes to the code. This required a comprehensive understanding of the dependencies among different components within different subsystems. Good tool support that assists in impact analysis of ripple effects would be helpful.

6. Related works

To evaluate evolvability, Ramil and Lehman proposed metrics based on implementation change logs [23] and computation of metrics using the number of modules in a software system [17]. Another set of metrics is based on software life span and software size [27]. In [26], a framework of process-oriented metrics for software evolvability was proposed to intuitively develop evolvability metrics and to trace the metrics back to the evolvability requirements based on the NFR framework [5]. However, they do not explicitly address the evolvability analysis at architectural level. The best known quality models e.g. McCall [20], Boehm [3], FURPS [10], ISO 9126 [12] and Dromey [9], do not explicitly address evolvability. An approach was described in [19] to measure software architecture's quality characteristics through identified key use cases, based on the customization of the ISO 9126 standard. An ontological basis which allows for the formal definition of a system and its change at the architectural level is presented in [24].

Kolb et al. [14] presented a case study in refactoring an existing software component for reuse in a product line using the PuLSE approach. Experiences of using various assessment techniques for software architecture evaluation were presented in [8], where scenario-based assessment, software performance assessment and experience-based assessment were addressed. The scenario-based methods such as ATAM [7] would require quite a number of evolvability scenarios (to address and cover each of the seven subcharacteristics); a more important limitation is that while scenarios are concrete anticipated events in the system life-time, evolvability might concern high-level business requirements at an abstract level which calls for some more general type of analysis to identify implications on software architecture and corresponding evolution path.

7. Concluding remarks

In this paper, we described an analysis of a complex industrial control system, driven by the need to improve its evolvability. A set of evolvability subcharacteristics were described from the case perspective: analyzability, architectural integrity, changeability, portability, extensibility, testability and domain-specific attributes. In addition, an architectural evolvability analysis method (designated as AREA method) was applied to the complex industrial system. The method made the architecture requirements, corresponding design decisions, rationale and architecture evolution path more explicit, better founded and documented, and the resulting documentation of refactoring improvement proposals was widely accepted by the involved stakeholders. The analysis results served as an input and blueprint to the implementation teams. We want to point out that the commitment from the organization to perform such a total restructuring of a large system signifies the importance of software evolvability.

The AREA method is presently being applied in another case within ABB, through which we plan to further refine and validate the method. Another aspect that we are considering is to apply the method to address evolvability explicitly in the early design phase of a new development effort, since software architecture that is capable of accommodating change must be specifically designed for change [13].

References

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

[2] Bennett, K., Rajlich, V., "Software Maintenance and Evolution: a Roadmap", the Future of Software Engineering, Anthony Finkelstein (Ed.), ACM Press, 2000.

[3] Boehm, B.W. et al., *Characteristics of Software Quality*, Amsterdam, North-Holland, 1978.

[4] Bosch, J., *Design and Use of Software Architectures – Adopting and Evolving a Product-Line Approach*, Addison-Wesley, 2000.

[5] Chung, L. et al., *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, 2000.

[6] Ciraci, S., Broek, P., "Evolvability as a Quality Attribute of Software Architectures", ERCIM Workshop on Software Evolution, 2006.

[7] Clements, P., Kazman, R. and Klein, M. *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2002.

[8] Del Rosso, C., "Continuous Evolution through Software Architecture Evaluation: a Case Study", Journal of Software Maintenance and Evolution: Research and Practice, 2006.

[9] Dromey, G., "Cornering the Chimera", IEEE Software (January): 33-43, 1996.

[10] Grady, R., Caswell, D., *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, NJ, PrenticeHall, 1987.

[11] Hofmeister, C., Nord, R., Soni, D., *Applied Software Architecture*, Addison-Wesley, 2000.

[12] ISO/IEC 9126-1. International Standard, Software Engineering: Product Quality, Part 1: Quality Model, 2001.

[13] Isaac, D., McConaughy, G., "The Role of Architecture and Evolutionary Development in Accommodating Change", Proceedings of NCOSE'94, 1994.

[14] Kolb, R., Muthig, D., Patzke, T. and Yamauchi, K., "Refactoring a Legacy Component for Reuse in a Software Product Line: a Case Study", Journal of Software Maintenance and Evolution: Research and Practice, 2006.

[15] Lago, P., van Vliet, H., "Explicit Assumptions Enrich Architectural Models", ICSE, 2005.

[16] Land, R., Crnkovic, I., "Software Systems In-House Integration: Architecture, Process Practices and Strategy Selection", Journal of Information and Software Technology, vol 49, nr 5, p419-444, Elsevier, September, 2006.

[17] Lehman, M.M, Ramil, J.F. et al., "Metrics and Laws of Software Evolution: The Nineties View", IEEE Computer Press, pp 20-32, 1997.

[18] Lientz, B., Swanson, E., *Software Maintenance Management*, Addison-Wesley, Reading, MA, 1980.

[19] Losavio, F. et al., "ISO Quality Standards for Measuring Architectures", the Journal of Systems and Software, 2004.

[20] McCall, J.A., Richards, P.K. and Walters, G.F., Factors in Software Quality, National Technical Information Service, 1977.

[21] Pei Breivold, H., Crnkovic, I. and Eriksson, P. J., "Analyzing Software Evolvability", Proceedings of the 32nd IEEE International computer Software and Applications Conference, 2008.

[22] Quecke, G., Ziegler, W., "Mesch - an approach to resource management in a distributed environment", In Proceedings of the First IEEE/ACM International Workshop on Grid Computing, Springer-Verlag, pp. 47–54, 2000.

[23] Ramil, J.F., Lehman, M.M., "Metrics of Software Evolution as Effort Predictors - A Case Study", Proceedings of ICSM, 2000.

[24] Rowe, D., Leaney, J., "Evaluating Evolvability of Computer Based Systems Architectures – an Ontological Approach", Proceedings of International Conference and Workshop on Engineering of Computer-Based Systems, 1997.

[25] Strauss, A. and Corbin, J. M., *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory* (2nd edition), ISBN 0803959400, Sage Publications, 1998.

[26] Subramanian, N., Chung, L., "Process-Oriented Metrics for Software Architecture Evolvability", Proceedings of IWPSE, 2002.

[27] Tamai, T., Torimitsu, Y., "Software Lifetime and its Evolution Process over Generations", Proceedings of ICSM, 1992.

[28] Weiderman, N.H. et al., "Approaches to Legacy Systems Evolution", Technical Report CMU/SEI-97-TR-014, 1997.

[29] Yang, H., Ward, M., *Successful Evolution of Software Systems*, Artech House Publishers, London, 2003.

Paper C

USING DEPENDENCY MODEL TO SUPPORT SOFTWARE ARCHITECTURE EVOLUTION

Hongyu Pei Breivold, Ivica Crnkovic, Rikard Land, Stig Larsson

Presented at the 4th International ERCIM Workshop on Software Evolution and Evolvability (Evol'08) at the 23rd IEEE/ACM Intl. Conf. on Automated Software Engineering
L'Aquila, Italy, September 2008

Abstract

Evolution of software systems is characterized by inevitable changes of software and increasing software complexity, which in turn may lead to huge maintenance and development costs. For long-lived systems, there is a need to address and maintain evolvability (i.e. a system's ability to easily accommodate changes) during the entire lifecycle. As designing software for ease of extension and contraction depends on how well the software structure is organized, this paper explores the relationships between evolvability, modularity and inter-module dependency. Through a case study of an industrial power control and protection system, we describe our work in managing its software architecture evolution, guided by the dependency analysis at the architectural level. The paper includes also the main analysis results, our experiences and reflections during the dependency analysis process in the case study.

1. Introduction

The role of software architecture in the evolution of software-intensive systems is being recognized and becoming increasingly important, as software architecture allows or precludes nearly all of the system's quality attributes [2, 11]. The evolution of software architecture implies integrating changing requirements and coping with stakeholders' concerns with respect to business, technology, process and organizational perspectives, which in turn may result in increased complexity. These phenomena of continuous change and increasing complexity in software systems were recognized by Lehman and expressed in his laws of software evolution [23]. In addition, one property of software systems noted by Brooks [5] is invisibility of software structure representation, which further negatively affects the software architecture evolution. Therefore, a lot of research has been done in exploring the relationship between the design of a complex system and the manner in which this system evolves over time [27]. We describe in our earlier work [34] an evolvability model which refines software evolvability into a collection of subcharacteristics that can be measured through a number of measuring attributes. This paper is a continuation of our earlier work [34] and further explores one particular measuring attribute, i.e. modularity, which affects the behavior of a design with respect to most of the evolvability subcharacteristics, as designing software for ease of extension and contraction depends on how well the software structure is organized and modular designs are argued to be more evolvable [27, 33], i.e. these designs facilitate making future adaptations. Although the value of modularity has been long recognized [41], not much data has been published with respect to large scale industrial software systems [22]. To enrich the knowledge in this direction, we describe our experiences through an industrial case study, with respect to (i) exploring the relationship between software evolvability, modularity and inter-module dependencies; (ii) using dependency model to support software architecture evolution; and (iii) to share industrial software evolution experiences with respect to reflections from the dependency analysis process.

The remainder of this paper is structured as follows. Section 2 summarizes our evolvability model and in particular explores the relationship between software evolvability, modularity and inter-module dependencies. Section 3 presents the methodology that we used in the case study. Section 4 presents the case study of an industrial control and protection software system and describes our work in managing the software architecture evolution through dependency analysis. Section 5 discusses the experiences we gained through the case study. Section 6 reviews related work and finally section 7 concludes the paper.

2. Evolvability, Modularity and Inter-Module Dependencies

This section summarizes first the evolvability model from our earlier work [34] and secondly, explores further the relationships between modularity, evolvability subcharacteristics and inter-module dependencies.

2.1. Evolvability Model

Software evolvability is a multifaceted quality attribute [35]. Based on the definition of evolvability in [35], analysis of various quality models [4, 13, 16, 21, 29], the software quality challenges and assessment [15], the types of change stimuli and evolution [9], and experiences we gained through industrial case studies, we have identified subcharacteristics that are of primary importance for an evolvable software system, and outlined a software evolvability model that provides a basis for analyzing and evaluating software evolvability. The idea with the evolvability model is to further derive the identified subcharacteristics to the extent when we are able to quantify them and/or make appropriate reasoning about the quality of service, as in Figure 1.

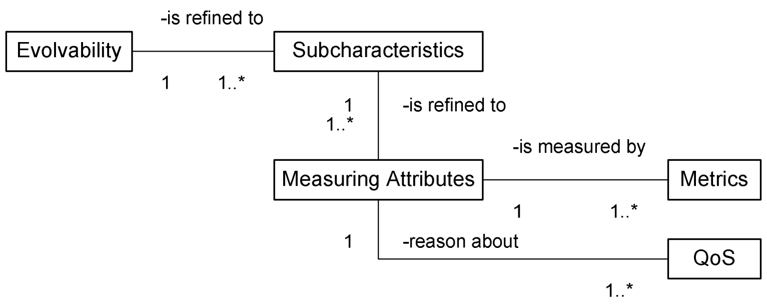


Figure 1 Elements of the evolvability model

The subcharacteristics and examples of their measuring attributes described in [34] are summarized in Table 1. Definitions of these subcharacteristics are provided in section 2.2. Failing in achieving any of these subcharacteristics probably will undermine the system's ability to be evolved.

Table 1 Subcharacteristics of evolvability and measuring attributes

Subcharacteristics	**Measuring Attribute**
Analyzability	modularity, complexity, documentation
Architectural Integrity	architectural documentation
Changeability	modularity, complexity, coupling, change impact, encapsulation, reuse
Extensibility	modularity, coupling, encapsulation, change impact
Portability	mechanisms facilitating adaptation to different environments
Testability	modularity, complexity
Domain-specific attributes	depend on the specific domains

2.2. Modularity and Subcharacteristics of Evolvability

This section explains the relationship between modularity and evolvability subcharacteristics. Modularity is a concept by which a piece of software is grouped into a number of distinct and logically cohesive subunits, presenting services to the outside world through a well-defined interface [12]. Modularization is a mechanism for improving the flexibility and comprehensibility of a system while allowing the shortening of its development time [32].

Modularity and analyzability Analyzability is the capability of the software system to enable the identification of influenced parts due to change stimuli, such as changes in environment, organization, process, technology and stakeholders' needs. Modularity plays an important role because an analysis of independent modules in isolation is easier to perform than in an analysis where a module is heavily dependent on other modules. Components that have excessive and unexpected dependencies are hard to work with because they cannot be understood easily in isolation. Statistics show that between 50% and 90% of software maintenance involves the understanding of the software being maintained [40], which implies the essence of modularity to achieve software analyzability.

Modularity and architectural integrity Architectural integrity is the non-occurrence of improper alteration of architectural information. A direct connection between modularity and architectural integrity does not exist. However, the modularization mechanisms and techniques, tactics and rationale for each design choice need to be documented to ensure architectural integrity. This documentation process is essential for the architecture to allow unanticipated changes in the software without compromising software integrity and to evolve in a controlled way [3].

Modularity and changeability Changeability is the capability of the software system to enable a specified modification to be implemented and avoid unexpected effects. Modularity plays an important role in software changeability because it reduces the probability that a change to one module propagates to other modules, and vice versa, to keep outside modifications from propagating into the module. According to [2], modularity increases the range of manageable complexity and accommodates uncertainty. Components that have excessive and unexpected dependencies are hard to work with because changes to functionality cannot be easily localized. Modularity determines software quality in terms of changeability [18].

Complex relationships between components make it difficult to anticipate and identify the ripple effects of changes [14].

Modularity and extensibility Extensibility is the capability of the software system to enable the implementation of extensions to expand or enhance the system with new capabilities and features with minimal impact to the existing system. Modularity plays an important role in extensibility because it supports separating concerns and enables definition of extension points [10] based on such considerations as coupling, cohesion. Components that have excessive and unexpected dependencies are hard to work with because the impact of extensions to functionality cannot be easily localized, and may adversely impact the capability of the software system to handle future additions without the need to rewrite existing functionality.

Modularity and portability Portability is the capability of the software system to be transferred from one environment to another. Modularity plays an important role in portability because it enforces information hiding behind a platform-independent interface, and ensures that the interface does not expose functions that are dependent on a particular platform.

Modularity and testability Testability is the capability of the software system to enable modified software to be validated. Modularity plays an important role in testability because it supports separating concerns among the parts of the system through coupling, cohesion and the likelihood of changes, so that different parts of the system can be tested separately without being interfered by each other. Monolithic characteristic in design may result in additional efforts in testing, as error corrections in one part of the software might require retesting of the other parts or the whole system. Having to link in many different libraries also leads to increased testing effort, particularly in the case of cyclic dependencies, where unit testing and releasing become difficult and error-prone.

Modularity and domain-specific attributes Domain-specific attributes are the additional quality subcharacteristics that are required by specific domains. The relationship between modularity and domain-specific attributes depends on the particular attribute and domain context. For instance, component exchangeability in the context of service reuse [26] is one domain-specific attribute within the distributed domain, e.g. wireless computing, component-based and service-oriented applications. In this context, modularity plays an important role because encapsulation mechanism shields the business logic and implementation from the outside world and thus enables component exchangeability.

2.3. Modularity and Inter-Module Dependency

Inter-module dependency is one of many indicators and measures for achieving modularity. Excessive inter-module dependencies have long been recognized as an indicator of poor software design [37]. They diminish the ability to reason about components of the software architecture in isolation. It becomes also difficult to assess and manage change impacts.

One way to visualize these dependencies is the Design Structure Matrix (DSM)[1], which is a representation and analysis mechanism for system modeling with respect to system decomposition and integration. Several architectural styles and dependency types, e.g. cyclic and hierarchical dependencies, are detectable in this matrix. There are two main categories of DSMs: static and time-based [6]. Static DSMs represent system elements and are analyzed with clustering algorithm. Time-based DSMs represent activity flows and are analyzed with sequencing algorithms. In this paper, we focus on static DSMs to reveal software structure problems during software evolution and explore alternative architectures to improve the evolvability of the software system.

3. Research method

We designed and conducted the dependency analysis of the control and protection system software which consists of more than one million lines of C and C++ code. The approach described in [37] was applied and we performed the following steps:

Step 1: Understand application and Dependency Structure Matrix representation.

Step 2: Create preliminary Dependency Structure Model of the application, using the hierarchical structure of the code's own namespace.

Step 3: Create conceptual architecture.

Step 4: Organize the Dependency Structure Model to reflect the intended conceptual architecture.

Step 5: Define design rules, specifying external library usage and application interdependencies.

[1] http://www.dsmweb.org

Step 6: Perform dependency management during software evolution.

Two potential parser alternatives were considered, i.e. Doxygen and Microsoft Browser (BSC). Doxygen was in the end not selected for analyzing and parsing the source files. The reason is that it does not correctly resolve dependencies when the symbol names are not unique, i.e. Doxygen can mix up a local variable reference for a global variable reference if they have the same name. It also has problems with symbol names used in multiple contexts. The BSC module was instead chosen to be used as input for generating the initial dependency model. It processes source code written in both procedural and object-oriented languages (e.g., C and C++), capture indirect calls (dependencies that flow through intermediate files), run in an automated fashion and output data in a format that could be input to a DSM. The BSC module analysis is file based and supports member level expansion of the files displayed in the dependency model.

We used Lattix[2], a source code level DSM derivation tool to extract code dependencies and examined the following kinds of dependencies:

Class reference: If class A refers to class B, e.g. as in an argument in a method, then A depends on B.

Invokes: If a function in class A calls to a function or a constructor of class B, then A depends on B.

Inherits: If class A is a subclass of class B, then A depends on B.

Data member reference: If a function in class A makes reference to a data member of class B, then A depends on B.

Three persons were actively involved in and performed the analysis process – one researcher from the research center, one software architect and one key software developer from the development unit of the analyzed system. The focus of the researcher was to apply the tool and analysis approach on the analyzed software system, attain an overview of the dependency situation and identify hotspots in the architecture and implementation. The software architect and the key software developer from the development unit have provided with information through daily meetings to make the conclusions objective. They also supported with their comprehensive

[2] http://www.lattix.com

domain knowledge, especially during the iterative process of creating a conceptual architecture for the analyzed system, where they identified the subsystems and modules in each layer. The risk of bias has been further decreased through the involvement of other researchers in the analysis of the experiences. The dependency analysis process took approximately three weeks. The architecture hotspots and refactoring solution proposals for the evolution path of the software system were identified. These proposals were discussed with the main technical responsible persons and architects, documented and transferred further to the implementation teams. Additionally, the experiences described in section 5.1 are summaries of the opinions of the involved stakeholders from the development unit.

4. Case study

The power control and protection system is built up from a basic system which handles communication, I/O and services, and from application functions that are combined to define various products. Software development is performed by several different development teams from two separate business units and across different geographical locations. We focused on the basic system which is the platform for different product types, i.e. control and protection as well as combinations of these.

The main problem with the original software architecture was the existence of tight coupling among components, which has led to additional work to modify some existing functionality and add support for new functionality in various products. This problem was discussed during the architecture workshops with the stakeholders, including people from product management, software architecture team and key software development team. Thus, inventory of candidates for modularization through dependency analysis was identified as the first top priority architecture requirement. Accordingly, the main focus of our case study was to analyze the software architecture in terms of inter-module dependencies, and to achieve a precise dependency overview for supporting software evolution. We identified potential flaws in architecture, implementation violations and defined an evolution path of the software architecture. In addition, we succeeded to convince the management of the effectiveness of using dependency model to guide and support software architecture evolution.

4.1. Examples of Analysis

We performed static software analysis using DSM models based on source code dependencies to extract dependency relations. Since the complete assessment of components cannot be presented due to space limitations, we select a subset and exemplify with two examples from the case to illustrate component evolution through inter-module dependency analysis. The examples are chosen to be understandable for people outside the power technology domain, while still representative and illustrative for the many various discussions and solutions that occurred during the analysis. The identified hotspots are analyzed in terms of the following views: (i) problem description: the problem and disadvantages of the original design of the component; (ii) requirements: the new requirements that the component needs to fulfill; (iii) improvement solution: the architectural solution to design problems; and (iv) rationale and architectural consequences: the rationale for design decisions and architectural implications of the deployment of the component.

4.1.1. Example 1 - Web Server

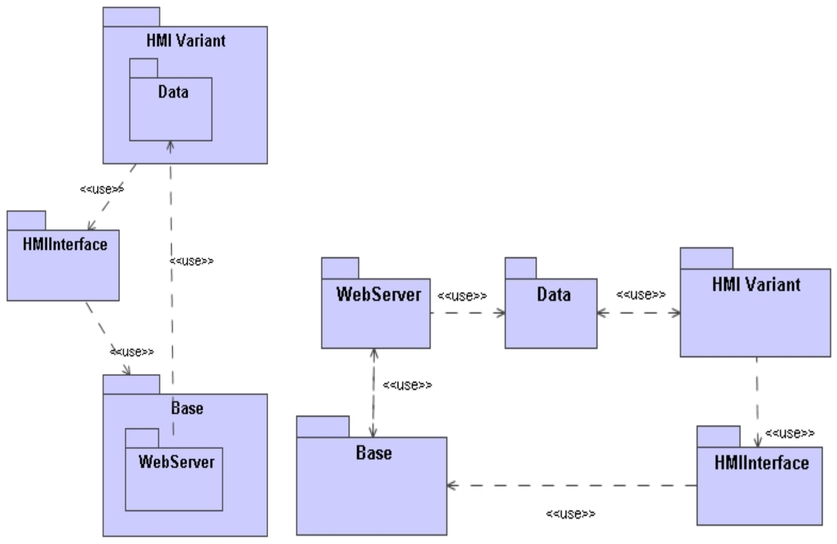
The *Web Server* subsystem is used to monitor the process and status of devices with respect to measurements, events and alarms. It consists of three main parts: a third-party software module, web client application and the software interface between client and server applications. The web client application is a combination of static and dynamic web pages, client-side scripts and style sheets.

Problem Description. Two cyclic dependency problems exist and these dependencies need to be removed, since we cannot change anything to either the module without possibly affecting the others. Accordingly, they prevent us from developing, testing or releasing modules independently.

(1) The *Web Server* subsystem existed within the *Base* system as shown in Figure 2a). It consists of third-party software, which is intertwined with the control and protection system's product family. As a result, the code size of *Base* increases, and the *Base* is affected by the third-party software because *Base* needs to be updated and recompiled once there is any update or change of the third-party software in the *Web Server* subsystem. However, simply moving *Web Server* outside *Base* creates a problem of cyclic dependencies between *Web Server* and *Base* as shown in Figure 2b). The dependency matrix in Figure 4a) illustrates also the cyclic dependencies between *Web*

Server and *Base*, i.e. the number in the first row indicates that *Base* uses *Web Server*, and vice versa as indicated by the number in the fourth row. Figure 4a) illustrates the dependencies among the components and visualizes the dependency violations, i.e. the implementation and architectural violations that are against design rules and design decisions. These violations are shown by the dependencies above the diagonal in the matrix (refer to [36, 31] for details). The numbers in the cells indicate the dependency strengths.

(2) The *Data* component encapsulated in *HMI Variant* subsystem is used by both the *HMI Variant* and the *Web Server* subsystem as shown in Figure 2a). To reduce the coupling between *Web Server* and *HMI Variant*, the *Data* component needs to be moved outside of *HMI Variant*. However, this creates another problem of cyclic dependencies between *HMI Variant* and *Data* as shown in Figure 2b). The dependency matrix in Figure 4a) illustrates also the cyclic dependencies between *Data* and *HMI Variant*, i.e. the number in the second row indicates that *Data* uses *HMI Variant*, and vice versa as indicated by the number in the third row.



a) b)

Figure 2. Conceptual view of the original correlations between Web Server and HMI components

Requirements. The *Web Server* must be isolated and moved outside *Base*. The *Data* component must be moved outside *HMI Variant*. In addition, the dependencies from *Base* to *Web Server*, as well as dependencies from *Data* to *HMI Variant* need to be removed.

Improvement Solution. The original architecture is transformed by partitioning the *HMI Variant* and *Base* respectively so that the cost for component modification is reduced. The revised conceptual architecture is illustrated in Figure 3.

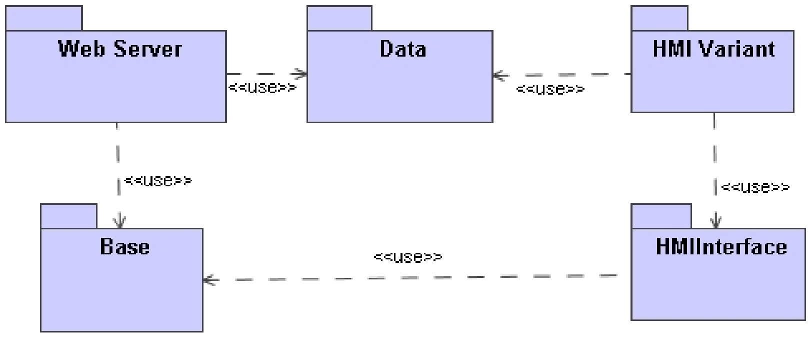
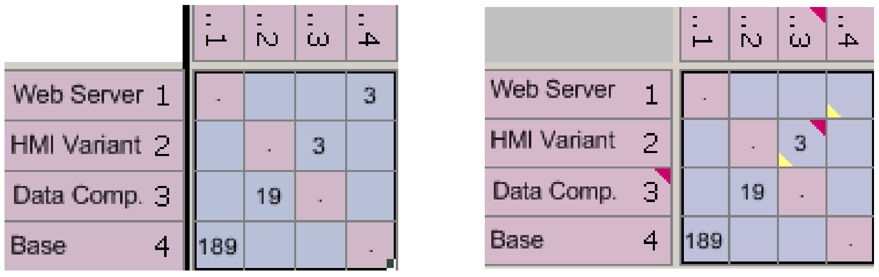


Figure 3. Conceptual view of the refactored correlations

Rationale and Architectural Consequences. The dependencies from *Web Server* to *Base* exist because some files in the *Web Server* component are used by the start-up sequence files in the *Base*. Accordingly, the implementations in the start-up sequence files were modified, and equivalent function was implemented in the application main module instead in order to remove the dependencies from *Base* to *Web Server* as illustrated in Figure 4b). In this process, we break the cyclic dependencies between *Web Server* and *Base* by moving the classes and functions that they both depend on into the application main module. The dependencies from *Data* to *HMI Variant* are caused by dead codes that are not in use any more.

The revised system architecture consists of a number of cohesive, modular subsystems and components with their implementations hidden behind well-defined interfaces. The probability that a change to one module (e.g. *HMI Variant* or *Web Server*) propagates to other modules is reduced.



a)

		1	2	3	4
Web Server	1	.			3
HMI Variant	2		.	3	
Data Comp.	3		19	.	
Base	4	189			.

b)

		1	2	3	4
Web Server	1	.			
HMI Variant	2		.	3	
Data Comp.	3		19	.	
Base	4	189			.

Figure 4. Dependencies before a) and after refactoring b)

4.1.2. Example 2 – Base

The *Base* software is used to provide a collection of services, as well as a platform that provides means of instantiation and configuration of application functions.

Problem Description. The *Base* software is a mixture of components that were traditionally implemented as function-oriented subsystems. They were not ordered according to any architectural styles. Direct connections and dependencies existed among components. If a change is made for a component, this implies changes to other components as well. The original coarse-grained architecture is depicted in Figure 5.

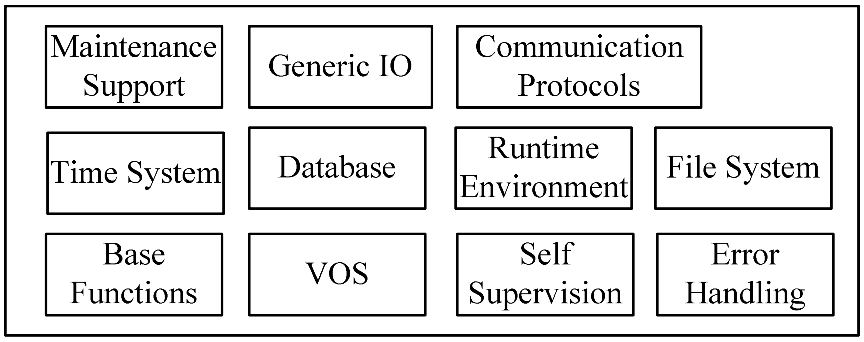
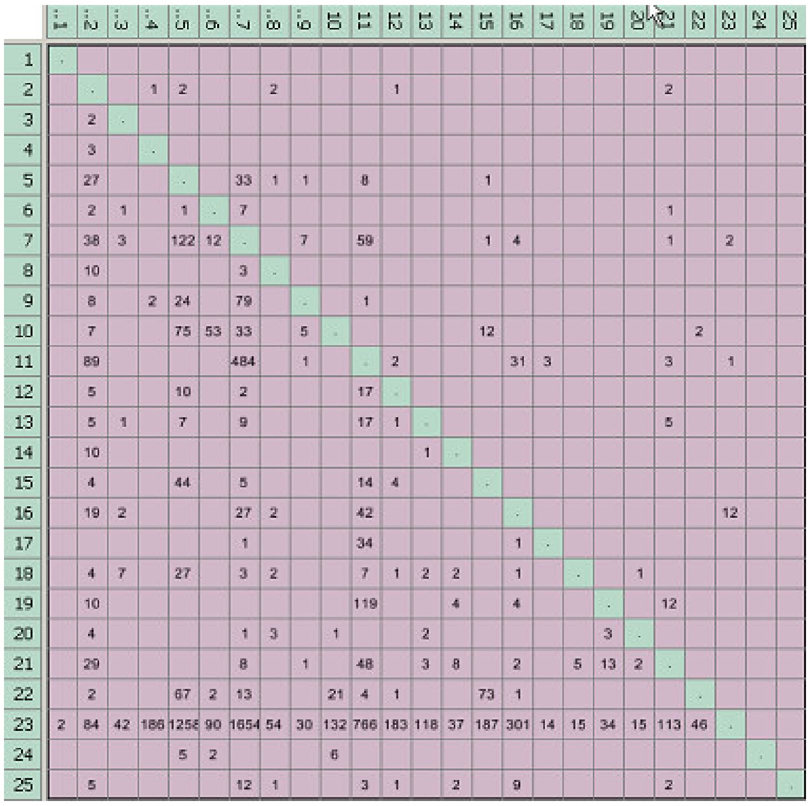


Figure 5. A conceptual view of the original software architecture

The initial DSM is created after loading the code base as in Figure 6.



	1:	2:	3:	4:	5:	6:	7:	8:	9:	10:	11:	12:	13:	14:	15:	16:	17:	18:	19:	20:	21:	22:	23:	24:	25:
1	.																								
2		.		1	2			2				1									2				
3		2	.																						
4		3		.																					
5		27			.		33	1	1		8				1										
6		2	1		1	.	7														1				
7		38	3		122	12	.		7		59				1	4					1		2		
8		10					3	.																	
9		8		2	24		79		.		1														
10		7			75	53	33		5	.					12							2			
11		89					484		1		.	2				31	3				3		1		
12		5			10		2				17	.													
13		5	1		7		9				17	1	.								5				
14		10											1	.											
15		4			44		5				14	4			.										
16		19	2				27	2			42					.							12		
17							1				34					1	.								
18		4	7		27		3	2			7	1	2	2		1		.		1					
19		10									119			4		4			.		12				
20		4					1	3		1			2						3	.					
21		29					8		1		48		3	8		2		5	13	2	.				
22		2			67	2	13			21	4	1			73	1						.			
23	2	84	42	186	1258	90	1654	54	30	132	766	183	118	37	187	301	14	15	34	15	113	46	.		
24					5	2				6														.	
25		5					12	1			3	1		2		9					2				.

Figure 6. Initial DSM for the code base

The x-axis and the y-axis of the matrix represent the same subsystems which are numbered sequentially. The dependencies for each subsystem are read down a column. Reading column 1, we see that subsystem1 depends on subsystem23 with dependency strength of '2'. This figure reveals the tight couplings among components and violations of design decisions (shown by the dependencies above the diagonal in the matrix).

Requirements. Clear boundaries between different parts of the system need to be defined. Late source code changes should not impose ripple effects through the system.

Improvement Solution. The revised conceptual architecture is illustrated in Figure 7. It consists of three layers including Utility layer, Middle Layer and Application Layer. The conceptual architecture was attained through an iterative process, i.e. daily discussions with the software architect and key software developer, with respect to what-if scenarios (what is the impact if we change) based on the dependency information provided by the inter-module dependency model.

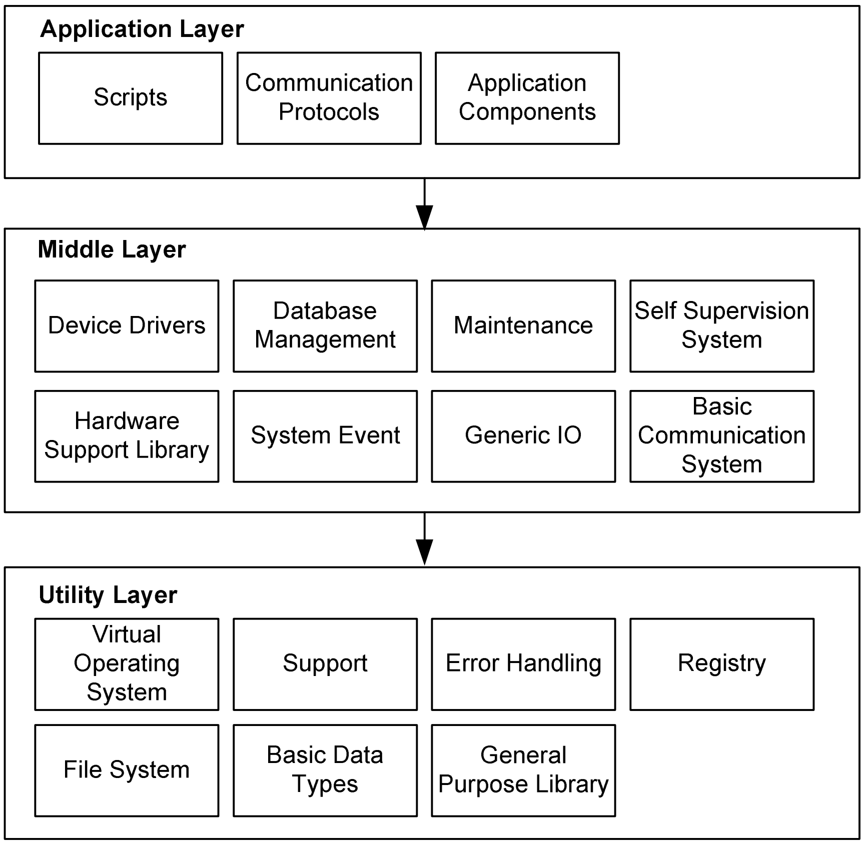
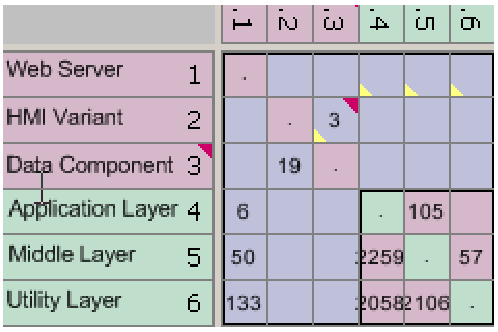


Figure 7. A conceptual architecture of the Base system

Rationale and architectural consequences. The original architecture is restructured into layered architecture, as the layers architectural pattern helps to structure applications to be decomposed into groups of subtasks at a particular level of abstraction [7]. The layered organization of software components offers a number of benefits such as reusability, changeability and portability [38]. In addition, cyclic dependencies across layers are identified as illustrated in Figure 8. For instance, reading column 6, we see that Utility layer depends on Middle layer with dependency strength of '57', indicating architectural layering violations.



		1	2	3	4	5	6
Web Server	1	.					
HMI Variant	2		.	3			
Data Component	3		19	.			
Application Layer	4	6			.	105	
Middle Layer	5	50			2259	.	57
Utility Layer	6	133			2058	2106	.

Figure 8. Dependencies after restructuring

The figure is a snapshot of the dependency model during the analysis process. The dependency violations are visualized by the dependencies above the diagonal in the matrix. As cyclic dependencies would make layers monolithic and inseparable, it is essential to break the cyclic dependencies. Two primary mechanisms [28] exist: (i) apply the dependency inversion principle; and (ii) create a new module or package, and move the classes that the cyclic dependent modules depend on into the new package.

5. Experiences and Reflections

This section presents firstly the benefits that were perceived by the involved stakeholders and secondly, our reflections through performing the inter-module dependency analysis.

5.1. Perceived Benefits of Performing Dependency Analysis Using Dependency Model

We summarize below visible benefits that were perceived and reported by the involved stakeholders in the organization.

a) It becomes easy to achieve a good overview of dependencies within the whole software system;

b) The software architects and software developers have increased potentials to do pre-studies in exploring different architectural and implementation solutions, due to the possibility of simulating changes in the dependency model without the necessity of making any modifications to the actual source code and due to the corresponding quick feedback on modifications from dependency analysis;

c) It enables a better and faster understanding of unfamiliar modules from dependency perspective; For instance, the development of *Web Server* subsystem was originally outsourced to another development unit located in another country. After the initial development, the original developers have changed their job and no one in the organization has the complete knowledge of the subsystem. However, the visualization of inter-module dependencies through the dependency model provides support for understanding the interaction of this subsystem with other parts of the system.

d) It facilitate discovery of implementation violation and perform quality check between various revisions; Design rules can be defined in the dependency model. Thus, it is possible to monitor if any implementation violations occur in the consecutive revisions to continuously check the quality of the architecture.

e) The possibility for reuse is increased; Excessive and unexpected dependencies reduce the reusability of components in different contexts and complicate the evolution of respective components, since each extension of components might affect other components. An example is managing inter-module dependencies in product line architecture. When a component is shared across multiple products, all components that this component depends on will also have to be shared or replicated in all of those products.

f) The time to do modularization work is shortened due to the quick visualization feedback from the dependency model.

5.2. Experiences and Reflections

We list below our reflections during the dependency analysis.

Gain management support Senior management generally has limited technical understanding to see the direct benefits of refactoring software architecture for improved quality, especially when there is a lack of economic models visualizing the benefits of investment. Although the software architects see the need for architecture restructuring, they usually do not have the roles of personnel resource management to execute the restructuring. In the case study, the three week dependency analysis succeeded to convince the management of the priority of architectural refactoring through the measure of dependency model. As a result, the management determined to continue with software architecture quality improvement activities instead of only focusing on providing functionalities.

Document rationale for each design decision Although the representation in the dependency structure matrix demonstrates the design decisions through the definition of design rules, e.g. the can-use and cannot-use rules, there is still a lack of explicit documentation of rationale behind the architectural decisions. Therefore, the dependency model needs to be complemented with design rationale information.

Apply routine dependency analysis as a quantitative indicator for judging the necessity of software refactoring and for supporting the choice of design decisions The software architecture needs to evolve to accommodate changes. Meanwhile, it is also essential to define design rules and monitor if any implementation violations occur during the software evolution process. Thus, we suggest routine dependency analysis as an integral part and quantitative indicator for continuously judging the necessity of performing software refactoring. In this sense, the process is close to the idea of agile software development in terms of continuous reengineering.

In addition, the choice of any design decisions can be supported by the quantitative measures from dependency analysis. It is a challenging task to make appropriate architectural decisions especially when there is a lack of quantitative measurement of the corresponding impacts on the system. Although there exist design tactics that assist in making design decisions, their corresponding impact within a particular system is still on an intuitive and qualitative level. Therefore, we suggest complementing with dependency analysis to better support design decisions, i.e. qualitatively reason about and quantitatively measure the impacts to make more accurate estimation on workload when making architectural changes.

Combine static code analysis with dynamic information extraction The case study shows that it is beneficial to perform static dependency analysis of source code to assist in software architecture evolution. Another aspect that is of interest is to identify and analyze the runtime structure and behavior of the software, and identify the runtime components and their dependencies. An example is to reconstruct software architectures in terms of pattern recognition. Patterns whose implementation involves dynamic mechanisms will require extraction of dynamic information [17]. This suggests a combination of extracting dynamic information of a system at run time and static source code analysis.

Combine different means for improved modularization In the case study, there have been discussions about techniques and means to increase modularization, as well as the potentials of combining different approaches for improved modularization and quality attributes. For instance, studies [20, 30] have shown that aspect-oriented software development can be applied in conjunction with object-oriented programming in order to achieve better modularity, reuse and adaptability in complex software systems [31]. As part of the dependency analysis process, we have identified some means for providing modularization (as shown in Table 2) to support software evolution and to provide one way to let some part of a system change independently of all other parts. A modularization technique benefits a design only when the potential changes to the design can be well encapsulated by the technique [8]. In the case study, the improved modularization was achieved through applying several design principles, e.g. separation of concerns, encapsulation boundaries and architectural coupling reduction, together with object-oriented software engineering and layered architecture style.

Table 2. Examples of Means to Increase Modularization

Means to Increase Modularization	**Examples**
Design Principles	Separation of concerns
	Information hiding
	Encapsulation boundaries
	Narrow component interfaces
	Architectural coupling reduction
Software Engineering Paradigms	Object-oriented software engineering
	Component-based software engineering
	Service-oriented software engineering
	Aspect-oriented software engineering
	Feature-oriented programming
Object-oriented Design Patterns	e.g. model-view-controller
Formal Specification	Specification of interfaces between components
	Assembling of components with compatible specifications
Programming Languages	e.g. coding guidelines for enabling modularization in programming languages
Modeling Techniques	Architectural description languages, e.g. ACME
	UML being enhanced with additional modularity mechanisms and abstraction, e.g. aspects, features
Architecture Styles	e.g. layer architectural style

6. Related work

The link between modularity and evolution was described by Simon [39] who argued that nearly-decomposable systems facilitate experimentation and problem solving. [22] examined the design evolution of one open source software product and one company software product platform through the modelling lens of design rule theory and design structure matrices. The idea of using design rules and DSM was similar to the way that we have performed in our case study. We further enrich the data with experiences and reflections through our dependency analysis of a complex industrial software system.

There exist different ways to visualize dependencies. [27] describes the concept of DSM and the application of design rules to identify violations, and to keep the code and its architecture in conformance with one another. Checking the conformance between design and implementation has been explored in [19]. Li [24] proposed object-oriented system dependency graph to calculate the impact of changes made to a class, with focus on three relationships, i.e. containment, use/reference and inheritance. Sullivan et al. [41] and Lopes et al. [25] have presented that DSM modeling can capture Parnas' information hiding criterion [32] and is valuable for software design. [1] formalizes this reasoning by showing that modularity creates design options.

The Architecture Tradeoff Analysis Method (ATAM) [2] is a method for evaluating software architectures in terms of quality attribute requirements to achieve better architecture. It is used to expose the possible areas of risks, non-risks, sensitivity points and trade-off points in the software architecture. Since it relies on the knowledge of the architect and has no provision for code inspection, it is not a precise instrument [2] as it is possible that some risks remain undetected. As a dependency model has the feature of being able to quantitatively and thus objectively visualize the inter-module dependencies, it can be used as a complementary approach to ATAM when there is existence of code.

7. Conclusions and future work

In this paper, we explored the links between evolvability, modularity, as well as inter-module dependency, and described a dependency analysis of a complex industrial power control and protection system, using the inter-module dependency model. The analysis was driven by the need of improving software evolvability, and it was performed by three persons (one researcher, one software architect and one key software developer), taking approximately three weeks. The purpose of the analysis is to visualize dependencies to provide direction to hotspots in the architecture and implementation. The resulting analysis documentation was widely accepted by the stakeholders involved in the analysis process and became a blueprint for further implementation improvement. Besides, the management was convinced of the effectiveness of using dependency model as a means to guide and support software architecture evolution. Additionally, the quantitative results also convinced them of the priority of improving architecture for better quality, instead of only focusing on functionality.

Our plans are to apply dependency model in new cases and in new domains, and further complement the static analysis with dynamic execution analysis. In addition, we need to consider the impact with respect to the software system's behavior, quality and any possible tradeoffs when we introduce any modularization mechanism and technique. Thus, another research area that is of interest is to investigate the impact of the choice of modularization mechanisms, as they might have consequences for flexibility and other concerns, such as runtime qualities, e.g. performance and scalability, etc.

References

[1] Baldwin, C. Y., Clark, K. B., Design Rules, vol 1, The Power of Modularity, MIT Press, 2000.

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

[3] Bennett, K., Rajlich, V., "Software Maintenance and Evolution: a Roadmap", the Future of Software Engineering, Anthony Finkelstein (Ed.), ACM Press, 2000.

[4] Boehm, B. W. et al., *Characteristics of Software Quality*, Amsterdam, North-Holland, 1978.

[5] Brooks, F. P. "No Silver Bullet", IEEE Computer, Vol. 20, No. 4. 1987.

[6] Browning, T. R., "Applying the Design Structure Matrix to System Decomposition and Integration Problems: A Review and New Directions", IEEE Transactions on Engineering Management, 2001.

[7] Buschmann, F. et al., Pattern-Oriented Software Architecture: A System of Patterns. Chichester, NY: Wiley, 1996.

[8] Cai, Y. and Huynh, S., "An Evolution Model for Software Modularity Assessment", Fifth International Workshop on Software Quality, 2007.

[9] Chapin, N. et al., "Types of Software Evolution and Software Maintenance", Journal of Software Maintenance and Evolution: Research and Practice, 2001.

[10] Clements, P., Bachmann, F., Bass, L. et al., *Documenting Software Architectures – Views and Beyond*, 2007.

[11] Clements, P., Kazman, R., Klein, M., *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2002.

[12] Developing Architecture Views. http://www.opengroup.org/architecture/togaf8-doc/arch/chap31.html, visited 2008.

[13] Dromey, G., "Cornering the Chimera", IEEE Software (January): 33-43. 1996.

[14] Feng, T., Zhang, J., Li, W., "Applying Change Impact Analysis and Design Metrics in CBR Based Software Design Improvement", Proceedings of ISCIT, 2005.

[15] Fitzpatrick, R. et al., "Software Quality Challenges", 26th International Conference on Software Engineering, 2004.

[16] Grady, R. and Caswell, D., *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, NJ, PrenticeHall, 1987.

[17] Guo, G. Y., Atlee, J. M., Kazman, R., "A Software Architecture Reconstruction Method", WICSA, 1999.

[18] Huynh, S. and Cai, Y., "An Evolutionary Approach to Software Modularity Analysis", 1st International Workshop on Assessment of Contemporary Modularization Techniques, 2007.

[19] Huynh, S., Cai, Y. et al., "Automatic Modularity Conformance Checking", ICSE, 2008.

[20] Improve modularity with aspect-oriented programming. http://www.ibm.com/developerworks/java/library/j-aspectj/, visited 2008.

[21] ISO/IEC 9126-1. International Standard, Software Engineering – Product Quality – Part 1: Quality Model, 2001.

[22] LaMantia, M. J., Cai, Y. et al., "Analyzing the Evolution of Large-Scale Software Systems using Design Structure Matrices and Design Rule Theory: Two Exploratory Cases", WICSA, 2008.

[23] Lehman, M., "Laws of Software Evolution Revisited", Software Process Technology, 5th European Workshop EWSPT, 1996.

[24] Li, L., Change Impact Analysis for Object-Oriented Software, PhD thesis, George Mason University, Virginia, USA. 1998.

[25] Lopes, C. V., Bajracharya, S. K., "An Analysis of Modularity in Aspect Oriented Design, Proceedings of AOSD, 2005.

[26] Lüer, C. et al., "The Evolution of Software Evolvability", Proceedings of IWPSE, 2001.

[27] MacCormack, A., Rusnak, J., Baldwin. C. Y., “The Impact of Component Modularity on Design Evolution: Evidence from the Software Industry”, HSB Working Knowledge, 2008.

[28] Martin,R., Acyclic Dependency Principle -Granularity. http://www.objectmentor.com/resources/articles/granularity.pdf, visited 2008.

[29] McCall, J. A., Richards, P. K., Walters, G. F., “Factors in Software Quality”, National Technical Information Service, 1977.

[30] Mens, T., Demeyer, S., Software Evolution, Springer, 2008.

[31] Padayachee, A., Eloff, J.H.P., “The Next Challenge: Aspect-oriented Programming”, Proceedings of the Sixth IASTED International Conference on Modelling, Simulation, and Optimization, 2006.

[32] Parnas, D. L., On the Criteria to be used in Decomposing Systems into Modules, 1972.

[33] Parnas, D. L., “Designing Software for Ease of Extension and Contraction”, IEEE Transactions on Software Engineering, 1979.

[34] Pei Breivold, H., Crnkovic, I., Eriksson, P., “Analyzing Software Evolvability”, Proceedings of COMPSAC, 2008.

[35] Rowe, D., Leaney, J., “Defining Systems Evolvability – a Taxonomy of Change”, Proceedings of the IEEE Conference on Computer Based Systems, 1998.

[36] Sangal, N., Expressing Software Architecture with Inter-module Dependencies, EclipseZone, http://www.eclipsezone.com/articles/lattix-dsm/, visited 2008.

[37] Sangal, N., Jordan, E., Sinha, V., and Jackson, D., “Using Dependency Models to Manage Complex Software Architecture”, OOPSLA, 2005.

[38] Sarkar, S., Rama, “A Method for Detecting and Measuring Architectural Layering Violations in Source Code”, 2006.

[39] Simon, Herbert A., “The Architecture of Complexity”, Proceedings of the American Philosophical Society 106: 467-482, repinted in idem, (1981) the Sciences of the Artificial, 2nd ed. MIT Press, Cambridge, MA, 193-229, 1962.

[40] Stoermer, C., O'Brien, L., Verhoef, C., “Moving Towards Quality Attribute Driven Software Architecture Reconstruction”, Proceedings of the 10th Working Conference on Reverse Engineering, 2003.

[41] Sullivan, K., Cai, Y., Hallen, B., Griswold, W. G., “The Structure and Value of Modularity in Software Design”, SIGSOFT Software Engineering Notes, 2001.

Paper D

COMPONENT-BASED AND SERVICE-ORIENTED SOFTWARE ENGINEERING: KEY CONCEPTS AND PRINCIPLES

Hongyu Pei Breivold, Magnus Larsson

Presented at the 33rd Euromicro conference on Software Engineering and Advanced Applications (SEAA), Component Based Software Engineering (CBSE) Track
Lübeck, Germany, August 2007

Abstract

Component-based software engineering (CBSE) and service-oriented software engineering (SOSE) are two of the most dominant engineering paradigms in current software community and industry. Although they have continued their development tracks in parallel and have different focus, both paradigms have similarities in many senses, which also have resulted in confusion in understanding and applying similar concepts or the same concepts designated differently. In this paper, we present a comparison analysis framework of CBSE and SOSE and analyze them from a variety of perspectives. We discuss as well the possibility of combining the strengths of the two paradigms to meet non-functional requirements. The contribution of this paper is to clarify the characteristics of CBSE and SOSE, shorten the gap between them and bring the two worlds together so that researchers and practitioners become aware of essential issues of both paradigms, which may serve as inputs for further utilizing them in a reasonable and complementary way.

1. Introduction

Today, designing and implementing a large scale and complex system has been a challenging task. Two of the most well recognized software engineering paradigms coping with this challenge are: component-based software engineering and service-oriented software engineering.

Component-based software engineering (CBSE) provides support for building systems through the composition and assembly of software components. It is an established approach in many engineering domains, such as distributed and web based systems, desktop and graphical applications and recently in embedded systems domains. CBSE technologies facilitate effective management of complexity, significantly increase reusability and shorten time to market. On the other hand, the growing demands for Internet computing and emerging network-based business applications and systems are the driving forces for the evolvement of service-oriented software engineering (SOSE). Service-oriented design utilizes services as fundamental elements for developing applications and software solutions. Service-oriented design technologies offer great feasibility of integrating distributed systems that are built on various platforms and technologies and further push focus on reusability and software development efficiency.

SOSE has evolved from CBSE frameworks and object oriented computing [16] to face the challenges of open environments. Therefore, CBSE and SOSE are similar to each other in many senses. Both use similar approaches and technologies. Both have software architecture as the common source and base. Meanwhile, both paradigms have continued with their development tracks in parallel and have different focus. Consequently, the mixture of similarities and specialized utilization of concepts in CBSE and SOSE have also resulted in confusion in understanding and applying concepts in a correct way. This may lead to less efficient utilization and combination of these paradigms. Furthermore, since both CBSE and SOSE can co-exist in enterprise systems and complement each other [17], any divided understanding and different interpretation of the terminologies would lead to less efficient combination and adaptation of these paradigms in future software development. For these reasons, it is important to clarify the concepts, principles and characteristics of CBSE and SOSE, shorten the gap between them and bring these worlds together so that researchers and practitioners can become aware of both sides. This clarification may serve as inputs to the subsequent investigation in how to take advantages of the strengths of these two paradigms, how to adapt and integrate the component-based and service-oriented technologies, concepts and their strengths so that both component-base and service-oriented software engineering can complement each other to the ultimate extent.

The goal of this paper is to provide a clarification framework of the component-based and service-oriented software engineering to avoid any misunderstandings and misuses. A brief discussion of reasonable utilization, combination and adaptation of the two paradigms is also outlined through looking into a set of research studies in how they have been used. These studies have exampled the benefits of improved quality attributes of software solutions through combining CBSE and SOSE. Since both paradigms are evolving rapidly, there exists increasing research interest in further exploration of their combination potentials. We contend that a good understanding of respective characteristics is a necessary step for this exploration.

The remainder of the paper is structured as follows. Section 2 presents overview of component-based and service-oriented software engineering. Section 3 gives a comparison analysis framework of the two paradigms from different perspectives, including key concepts and principles, process, technology and composition. Section 4 discusses state of the art research in combining the strengths of CBSE and SOSE. Section 5 concludes the paper.

2. Overview of component-based and service-oriented software engineering

Component-based software engineering (CBSE) is a software engineering paradigm that aims to accelerate software development and promote software reusability and maintenance through assembling components to software systems that meet certain business requirements. The prerequisite requirements that enable components to be integrated and work together are component models and component framework [20]. Component models specify the standards and conventions that components need to follow during component composition and interaction. Component framework provides design time and run time infrastructure.

Numerous component models exist nowadays. Some examples are COM/DCOM/COM+, .Net component model, JavaBeans, Enterprise JavaBeans and CORBA component model. Examples of component models that have been developed specifically for applications to embedded systems include Koala [11], Rubus [21], PECOS [18].

Important areas of research within CBSE include, but not limited to, determination and specification of QoS (Quality of Service), predictability of non-functional properties, component interference and process related activities such as component classification, identification and selection, component adaptation, testing and deployment techniques.

Although CBSE has proved to be successful for software reuse and maintainability, it does not address all of the complexities software developers are facing today, such as varying platforms, varying protocols, various devices, the Internet, etc [7]. Service-oriented software engineering paradigm has emerged to address these issues.

Service-oriented software engineering (SOSE) is a software engineering paradigm that aims to support the development of rapid, low-cost and easy composition of distributed applications even in heterogeneous environments [13]. It utilizes services as fundamental elements for developing applications and solutions.

Important areas of research within SOSE include service foundations, service composition, service management and monitoring and service-oriented engineering [13]. Service foundations provide service-oriented communication technologies to support run time service-oriented infrastructure and connect heterogeneous systems and applications. These communication technologies provide the communication mechanisms between service providers and service requesters; they differ with respect to service description techniques and messaging functions [6]. Service composition encompasses necessary roles and functionality to support service composition [13]. The dynamic composition feature in SOSE makes QoS a major challenge. Different initiatives have emerged such as orchestration and choreography. Service management encompasses the control and monitoring of SOA-based applications throughout life cycle.

A key element in SOSE is the service-oriented interaction pattern, i.e. service-oriented architecture (SOA), which enables a collection of services to communicate with each other. SOA is a way of designing a software system to provide services to applications or other services through published and discoverable interfaces. The basic elements of service-oriented architecture are illustrated in Figure 1.

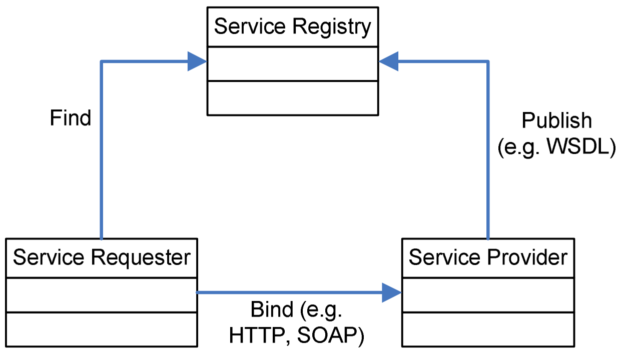


Figure 1. Service-oriented interaction pattern

As shown in Figure 1, SOA has three main actors: a service provider, a service requester and a service registry. The service provider defines service descriptions of a collection of services, supplies services with functionalities and publishes the descriptions of the services so as to make the services discoverable. The service registry contains service descriptions and references to service providers and provides mechanisms for service publishing and discovery [14], e.g. Universal Description, Discovery and Integration (UDDI). The service requester is a client that calls a service provider. It can be an end-user application or other services. A service requester searches in the service registry for a specific service via the service interface description. When the service interfaces match with the criteria of the service requester, the service requester will use the service description and make a dynamic binding with the service provider, invoke the service and interact directly with the service.

3. Classification of component-based and service-oriented software engineering

The main concepts and principles of CBSE and SOSE may look similar at the first sight, but differences exist in mechanisms, approaches and implementations. Therefore, we group particular characteristics that have similar concerns to describe the same or related aspects of CBSE and SOSE. The categories in the comparison framework that we are going to address are: key concepts and principles, process concerns, technology concerns, quality and composition.

3.1. Key concepts

A summary of the key concepts in CBSE and SOSE is listed in Table 1.

Table 1. Comparison of key concepts in CBSE and SOSE

Concepts	CBSE	SOSE
Module	Component	Service
Specification	Component contract	Service description
Interface	Component interface	Service interface
Assembly	Component composition	Service composition

3.1.1. Module

In CBSE, components are the building blocks that can be deployed independently and are subject to composition by third party [4]. Based on the formulation by Clemens Szyperski [15], a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. It can be both fine-grained providing specific functionality and coarse-grained encompassing complicated logics.

In SOSE, services are the building blocks that can be reused and offer particular functionalities. They are generally implemented as coarse-grained discoverable software entities [2], operating on larger data sets, encapsulating business functionality and exposing the functionality to any source that requests the functionality through well-defined interfaces. Thus, the services can be reused and accessed at various levels of the enterprise application and even across enterprises boundaries.

3.1.2. Specification

In CBSE, the component specification provides for the clients the definition of the component's interface, i.e. the operations and context dependencies. Furthermore, an abstract definition of the component's internal structure is specified for the component providers [4].

In SOSE, the service description is a service contract that advertises the following information: (i) service capabilities - stating the conceptual purpose and expected results of the service; (ii) interface - describing the service signatures of a set of operations that are available to the service requester for invocation; (iii) behavior - describing the expected behavior of a service during its execution; and (iv) quality - describing important functional and non-functional service quality attributes [12].

3.1.3. Interface

Although both CBSE and SOSE are interface-based in the sense that interfaces are the specifications of access points, the separation between service descriptions and service implementation is more explicit than the separation between component specification and implementation.

3.1.4. Assembly

In CBSE, component composition is the process of assembling components using connectors or glue code to form an assembly, a larger component or an application. The components are assembled through the component interfaces and the composition is made out of several component instances that are connected and interact together.

In SOSE, the composite services are built by composing service descriptions. The realization of the service composition is during run time when the service providers are discovered and bound.

3.2. Key principles

A summary of the key principles of implementation in CBSE and SOSE is listed in Table2.

Table 2. Comparison of key principles of implementation in CBSE and SOSE

PRINCIPLES	CBSE	SOSE
Coupling	Loose and tight coupling	Loose coupling
Self describing	Component specification	Service descriptions
Self contained	yes	yes
State	Stateless/stateful	Stateless/stateful
Location transparency	In some component models e.g. DCOM	yes

3.2.1. Coupling

CBSE enables both loose coupling and tight coupling. As a component is used within the scope of a component model, it needs to conform to the rules specified by the component model. A component model often uses one particular interaction style, such as broadcasting, asynchronous connection and connection-oriented style. All these interaction styles imply some kind of coupling between components, such as referential coupling and temporal coupling.

In contrast to CBSE, SOSE enables only loose coupling, with minimized dependencies between service providers and service requesters. The service providers need not to know anything about the service requesters or any other services. They have great flexibility in choosing their design and deployment environment to offer their services. Likewise, the service requesters or calling applications need not to know anything about underlying logic of the service implementation and service deployment except the service descriptions. The service descriptions are the only communication channel between service requesters and service providers. Service loose coupling is enabled through the use of service descriptions that allow services to interact within predefined parameters [5].

3.2.2. Self describing

Both CBSE and SOSE share the same self describing characteristic with their own specialization. In CBSE, the component specification is the key to the component's self describing characteristic and specifies the rules that the components must conform to.

In SOSE, the service description is the key to the service's self describing characteristic. The service provides its clients with all the relevant information in the service descriptions, which contain combinations of syntactic, semantic and behavioral information.

3.2.3. Self contained

In CBSE, components can be self contained. For example, for CCM, a component is 'a self-contained unit of software code consisting of its own data and logic, with well-defined connections or interfaces exposed for communication. It is designed for repeated use in developing applications; either with or without customization' [22].

In SOSE, services are self contained. The services provide the same functionality regardless of the other services, even if any other services may fail for some reason.

3.2.4. Stateless

Both components and services can be stateful or stateless. In SOSE, stateless services are used to meet the performance requirements and in some circumstances, the stateless property is optimal for services' reusability. As a result, the services should minimize the amount of state information they manage and the duration for holding the message information. Otherwise, the services would not be able to timely correspond to other service requesters. On the other hand, there are circumstances when stateful services are necessary so as to maintain states across several method calls by the same service requester. The service object creation policy determines whether a stateful service can be returned.

3.2.5. Location transparency

In CBSE, some component models can provide location transparency, e.g. DCOM allows component-based applications to be distributed across memory spaces or physical machines using proxies and stubs.

In SOSE, since services have their descriptions and location information stored in the service registry through e.g. UDDI, which is accessible to a variety of service requesters, services can be invoked by service requesters from different locations.

3.3. Development process concerns

Three aspects related to development process are identified for further comparison.

3.3.1. Building from pre-existing entities (components or services)

The main idea for CBSE is to build systems from pre-existing components. This feature applies in the same way for SOSE in the sense that systems can be built from composing appropriate pre-existing services to meet certain business functionality.

3.3.2. Separation of development process of system and entities (components or services)

In CBSE, the development process of component-based systems is separated from the development process of components. This feature applies in the same way for SOSE in the sense that services can be developed by various service providers across organizational boundaries and the service requesters need only to discover and invoke the services.

3.3.3. Development process

In CBSE, engineering a component-based software system is a process of finding components, evaluating and selecting proper components, testing, adapting if necessary and integrating the components into the software system, e.g. in the COTS-based development process. In SOSE, engineering a service-oriented computing system is a process of discovering and composing the appropriate services to satisfy a specification [8]. The process of service discovering, matching, planning and composing is essential. Service-oriented engineering process focuses more on run-time activities, such as dynamically adding, discovering and composing services illustrated in Figure 2.

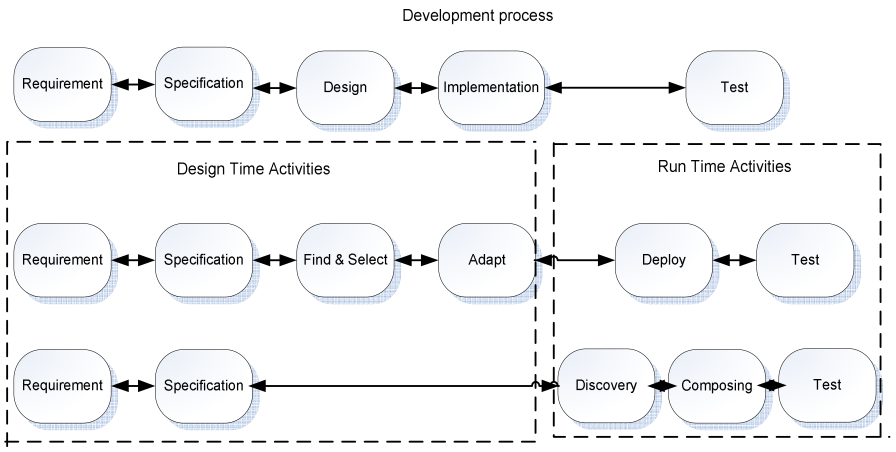


Figure 2. Comparison of typical activities during development process in CBSE and SOSE

3.4. Technology concerns

Three aspects are identified for further comparison: technology neutrality, encapsulation and static or dynamic behavior.

3.4.1. Technology neutrality

In CBSE, components need to conform and follow the rules that are set up by a specific component model. As a result, the feasibility to compose components of different component models is relatively limited. On the other hand, compliance to a certain technology may also lead to advantages in the sense that many solutions can be optimized since they can be directly supported by the specific technology.

In contrast to CBSE, SOSE provides the feasibility for services to be implemented in diverse technologies and for multiple applications running on different platforms to communicate with each other. This feasibility is enabled through applying commonly accepted message standards for interface descriptions to the services. Hence, the enterprise applications or solutions can cut across technology and platform boundaries, performing business functionalities by composing services from different sources of service providers.

3.4.2. Encapsulation

Encapsulation means that the business logic and implementation are shielded from the outside world. CBSE supports a variety of encapsulation types, ranging from white box exposing all the implementation, or gray box exposing parts of component implementation to black box. In the cases of white box and gray box, the component clients have the flexibility to make modifications to the component in order to meet specific needs in their solutions.

In contrast to CBSE, SOSE supports only black box encapsulation. The logical view of a service consists of one or a set of service interfaces and service implementation. A service can be regarded as a business logic entity which can be accessed and executed through the well-defined and formal interfaces by any service requester that wants to use the service. This is called the service interface level abstraction [5], which enables the services to act as black boxes, leading to the inflexibility of service requesters to modify services.

3.4.3. Static vs. dynamic

Two aspects are concerned:

(1) Binding

There are two types of binding: early binding and late binding. Early binding allows clients to obtain compile-time type information from the component's type library. Late binding allows clients to bind to components at run time and the compiler has no clue during build time about the method calls that are to be made at run time.

CBSE allows static early binding and supports dynamic late binding in some component models. An example is early and late binding to COM components. In early binding, the components are instantiated as needed and invocations of operations are based on the interface definitions, statically checked and bound to by the compiler. In late binding, components are bound by invoking IDispatch methods in COM that redirects dynamically to the sought interface. The choice of static or dynamic binding has both pros and cons, and consequently need to be taken into consideration during design. Static binding between components may lead to the disadvantage of less flexibility in facilitating changes, but it allows for stronger type checking during compile time and is much faster than the late binding approach.

SOSE allows only dynamic binding. The service requesters make targeted named calls and search in the service registry for a specific service. When the service requesters find the services that match certain criteria, the service requester will use the service description to make a dynamic binding with the service provider.

(2) Dynamic discovery and availability

Discovery implies the ability that an entity (component or service) is discovered for use. Availability is the ability that an entity (component or service) is operational or accessible when required for use. In CBSE, dynamic discovery and dynamic availability of components are not the major concerns [3].

In SOSE, services exhibit the feature of dynamic availability, since they can be added or removed from the service registry at any time. Consequently, services are readily available running entities and need to be dynamically discovered and composed in run time.

3.5. Quality concerns

Quality attributes can be classified into life cycle properties and run time properties. Hundreds of quality properties exist and we can not analyze all of them. Therefore, we choose only quality attributes that are of common or related interest to CBSE and SOSE.

3.5.1. Reusability as life cycle property

CBSE emerged to accelerate reusability of software. However, there are some constraints in achieving component reusability, such as component specification should be explicit, no architectural mismatches among composed components, etc.

Similar to CBSE, services can be reused to construct applications. In SOSE, the concern in having similar architecture needs not to be taken into consideration because of the technology neutrality, platform independence and interoperability characteristics of SOSE. On the other hand, extra emphasis is put on having explicit service descriptions.

There are several factors that contribute to the reusability of components and services. Firstly, both components and services are composable. This implies that the level of granularity of components and services need to be considered when taking reusability into account. The design of operations should be in a standardized manner and with appropriate level of granularity [5] so that the components or services can be reused and composed. Secondly, the separations between component/service development and applications also promote component and service reusability.

Recently, researchers have been active in investigating the possibilities of enhancing service reusability with service-oriented architectures. One study is presented by Zhu in [19], where he proposed the idea that services are new types of components and service-oriented architectures may provide more chances for the development of reusable components.

3.5.2. Substitutability as life cycle property

Substitutability means that alterative entity (component or service) implementation may be used with the constraints that the system can still meet the requirements on functional level and non-functional level. According to [15], white box and gray box reuse very likely prevents the component substitutability. In such cases, explicit conventions about the implementation information and changes that are made in components are required to achieve substitutability [4].

In SOSE, since the service-oriented interaction pattern enables the loose coupling characteristic between a service requester and service providers, services can be substituted with new services as long as the service descriptions fulfill the criteria from service requesters.

3.5.3. Interoperability as runtime property

The main idea in CBSE is to assemble components together to perform certain functionality. However, each component conforms to a certain component model that specifies different rules from another component model. Therefore, interoperability between heterogeneous components is still a challenging issue in CBSE. Although in some circumstances, interoperability can be achieved through implementing wrapper class or proxies.

On the other hand, broad interoperability among different vendors' applications and solutions can be achieved in SOSE through the use of well accepted standards. For instance, WSDL, UDDI, SOAP, XML [23]. These descriptions are independent of underlying platform, programming languages and implementation details and therefore promote interoperability.

3.6. Composition concerns

Three aspects are concerned.

3.6.1. Heterogeneous vs. homogeneous composition

In CBSE, components can only be assembled according to the rules specified by a specific component model; there is not much feasibility to assemble components that conform to different component models.

In SOSE, services which access and combine information and functions from different sources of service providers can be assembled into composite services to perform particular tasks [12]. The service-oriented software engineering principles, such as services are platform independent and loosely coupled, offer the feasibility that services from different sources of service providers can be used in the same composite service.

3.6.2. Design time/run time composition and composition mechanisms

In CBSE, components can be composed at design time and run time. Design time composition allows for optimization [4]. A component detaches its interface from its implementation, and conceals its implementation details, hence permitting composition without need to know the component implementation details [1]. The mechanisms for component composition vary from method calls, to pipes and filters or event mechanism [4]. Furthermore, component models provide also general architecture and mechanism for component composition. For example, component models require components to support introspective operations to enable component composition at assembly time or run time [17], e.g. the functionality and properties of the components can be discovered and utilized automatically at assembly time or run time.

In SOSE, services are composed at run time. Several mechanisms exist to compose services, such as pipe and filter which can direct the output of one service into the input of another service, orchestration and choreography. Orchestration utilizes a high-level scripting language to control the sequence and flow of service execution. It describes the behavior and interactions of a specific service provider with other involved services. BPEL4WS (Business Process Execution Language for Web Services) and WSCI (Web Service Conversation Interface) are examples of web service orchestration languages. Choreography describes the interactions between service providers that are collaborated for achieving business functionality. WS-CDL (Web Service Choreography Description Language) [24] is one example of choreography languages.

3.6.3. Predictability

In CBSE, the predictability of non-functional properties of the composition components from the properties of components remains to be a challenging issue. However, compared with SOSE, the use of static binding in CBSE may provide to a certain extent better predictability because of the clarification of interface-based design during assembly time.

To some extent, SOSE faces even more challenges in predictability because of its dynamic discovery and dynamic availability behaviors. Some of the examples of the challenges include how to predict the quality of service when services are discovered and invoked dynamically during run time, how to predict the quality properties when services are composed at run time? These are still interesting open research issues.

Based on the above comparison analysis, the main similarities and differences between CBSE and SOSE are summarized in Table 3.

Table 3. Summary of similarities and differences of CBSE and SOSE

	CBSE	SOSE
Process	Building system from pre-existing components. Separate development process of components and system. More activities involved in design time	Building systems from pre-existing services. Separate development process of services and system. More activities involved in run time
Technology	Constrained by component models. Ranging from white box, gray box to black box. Static and dynamic binding between components. Dynamic discoverability is not a major concern	Platform independency. Black box. Only dynamic binding between services. Dynamic discoverability
Quality	Interoperability concern between heterogeneous components. Achieve component substitutability through explicit specifications. Better predictability	Interoperability through universally accepted standards. Achieve service substitutability through service descriptions. Predictability issue
Composition	Homogenous composition. Design time and run time composition and design time composition allows for optimization. Pipe and filter; event mechanism etc. Composition is made out of several component instances	Heterogeneous composition. Services are composed at run time. Pipe and filter; orchestration etc. Composite services are built by composing service descriptions

4. Discussions

Because of the diverse nature of software systems, it is unlikely that systems will be developed using a purely service or component-based approach [10]. Therefore, the ability to combine the strength of CBSE and SOSE and use them in a complementary manner becomes essential. So far, a lot of research has been done in combining the strength of CBSE and SOSE for improved quality attributes of software solutions. Jiang and Willey proposed a multi-tiered architecture [9] that offers flexible and scalable solutions to the design and integration of large and distributed systems, where the architecture makes use of both services and components as architectural elements, offering flexibility and scalability in large distributed systems and meanwhile remaining the system performance. Wang and Fung [17] proposed an idea of organizing enterprise functions as services and implementing them as component-based systems in order to offer flexible, extensible and value-added services. Cervantes and Hall [3] addressed introducing service-oriented concepts into component model to provide support for late binding and dynamic component availability in component models. Since CBSE and SOSE keep on developing rapidly, exploring their combination potentials is still one interesting research topic.

5. Summary

In this paper, we have presented a comparison framework for component-based and service-oriented software engineering and discussed briefly the research efforts that have been done in combining the strengths of CBSE and SOSE for improved quality attributes.

An explicit clarification of the concepts, principles and characteristics of CBSE and SOSE is the first necessary step before further exploration in efficient utilization and reasonable combination of them in future applications. Discussions on state of the art research with respect to how to combine the two technologies in a complementary way can be helpful for further investigation of the long term advantages in introducing service-oriented architecture into component-based development, and integrating component-based and service-oriented architecture to offer added value in system development.

6. References

[1] Aoyama, M., “New Age of Software Development: How Component-Based Software Engineering Changes the Way of Software Development”, Proceedings of the first workshop on Component Based Software Engineering, 1998.

[2] Brown, A., Johnston, S. and Kelly, K., “Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications”, A Rational Software White Paper, 2002.

[3] Cervantes, H. and Hall, R. S., “Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model”, 2004.

[4] Crnkovic, I. and Larsson, M., *Building Reliable Component-Based Software Systems*, Artech House Publishers, 2002.

[5] Crnkovic, I., Larsson, S. and Chaudron, M., “Component-Based Development Process and Component Lifecycle”, Information Technology Interface, 2005.

[6] Dijkman, R. M. et al, “The State of the Art in Service-Oriented Computing and Design”, 2003.

[7] Hashimi, S., "Service-Oriented Architecture Explained", http://www.ondotnet.com/, 2003.

[8] Huhns, M. N. and Singh, M. P., "Service-Oriented Computing: Key Concepts and Principles", IEEE Internet Computing, Service-Oriented Computing Track, 2005.

[9] Jiang, M. and Willy, A. "Architecting Systems with Components and Services", Information Reuse and Intergration, 2005.

[10] Kotonya, G., Hutchinson, J. and Bloin, B., "A Method for Formulating and Architecting Component and Service-Oriented Systems", http://scse.comp.lancs.ac.uk/pubs/KotonyaHutchinsonBloin_SOSEBook.pdf, visited 2007.

[11] van Ommering, R., van der Linden, F. and Kramer, J., "*The koala component model for consumer electronics software*", In IEEE Computer, pages 78–85. IEEE, March 2000.

[12] Papazoglou, M. P., "Service-Oriented Computing: Concepts, Characteristics and Directions", Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE), 2003.

[13] Papazoglou, M. P., Traverso, P., Dustdar, S. and Leymann, F., "Service-Oriented Computing Research Roadmap", 2006.

[14] Stojanovic, Z. and Dahanayake, A., *Service-Oriented Software System Engineering: Challenges and Practices*, Idea Group, U.S, 2004.

[15] Szyperski, C., *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, 2002.

[16] Tsai, W. T., "Service-Oriented System Engineering: A New Paradigm", Proceedings of the 2005 IEEE International Workshop on Service-Oriented System Engineering (SOSE), 2005.

[17] Wang, G. and Fung, C. K., "Architecture Paradigms and Their Influences and Impacts on Component-Based Software Systems", Proceedings of the 37th Hawaii International Conference on Systems Sciences, 2004.

[18] Winter, M., Zeidler, C., Stich, C., "The PECOS Software Process", Workshop on Components-based Software Development Processes, ICSR 7 2002.

[19] Zhu, H., "Building Reusable Components with Service-Oriented Architectures", Information Reuse and Integration, 2005.

[20] Component-Based Design and Integration Platforms, http://www.artist-embedded.org/, 2002.

[21] Arcticus Systems, Rubus component model, http://www.arcticus-systems.com

[22] OMG. CORBA Components. Report ORBOS/99-02-01.

[23] W3C. World-Wide-Web Consortium: XML, SOAP, WSDL, http://www.w3c.org/

[24] W3C World Wide Web Consortium, Web Services Choreography Working Group, http://www.w3.org.

Paper E

MIGRATING INDUSTRIAL SYSTEMS TOWARDS SOFTWARE PRODUCT LINES: EXPERIENCES AND OBSERVATIONS THROUGH CASE STUDIES

Hongyu Pei Breivold, Stig Larsson, Rikard Land

Presented at the 34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Software Process and Product Improvement (SPPI) Track
Parma, Italy, September 2008

Abstract

Software product line engineering has emerged as one of the dominant paradigms for developing variety of software products based on a shared platform and shared software artifacts. An important and challenging type of software maintenance and evolution is how to cost-effectively manage the migration of legacy systems towards product lines. This paper presents a structured migration method and describes our experiences in migrating industrial legacy systems into product lines. In addition, we present a number of specific recommendations for the transition process which will be of value to organizations that are considering a product line approach to their business. The recommendations cover four perspectives: business, organization, product development processes and technology.

1. Introduction

Today, technical, business and environment requirements change at a tremendous speed [2]. The ability to launch new products and services with major enhancements within short timeframe has become essential for companies to keep up with new business opportunities. The need for differentiation in the marketplace, with short time-to-market as part of the need, has put critical demands on the effectiveness of software reuse. In this context, software product line approach has become one of the most established strategies for achieving large-scale software reuse and ensuring rapid development of new products [4]. However, product line development seldom starts from scratch. Instead, it is very often based on existing legacy implementations [14], as legacy systems represent substantial corporate knowledge and investment [Tilley 1999]. These legacy systems are usually critical to the business in which they operate [Ransom et al. 1998]. Therefore, they are maintained and evolved to fit existing and expanding markets and customer needs. However, not much data has been published with respect to experiences and lessons learned in product line migration [21]. To enrich the knowledge in this direction, we describe our experiences and observations through two industrial case studies, with respect to (i) migrating legacy systems to product line architecture, and (ii) observations with respect to business, organization, process and technology perspectives during product line transition process. The contribution of this paper is to provide experiences through industrial examples in product line migration that can be shared within the software industry, and can enable future application and utilization of the product line concept to be additionally efficient and effective.

The remainder of this paper is structured as follows. Section 2 describes the research method and the context of the two industrial cases including the motivations for product line migration. In section 3, we present the migration method that we applied in the transition process and exemplify with one case to demonstrate the usage of the method. Section 4 discusses our observations and recommendations made in the two case studies, with respect to business, organization, development processes and technology perspectives. Section 5 reviews related work and section 6 concludes the paper.

2. Research method

This research is based on two industrial cases. The first two authors took part in the development of a product line architecture in both cases. All experiences are thus first-hand; in addition, other participants in the cases have provided us with material to make the conclusions less subjective. The risk of bias has been further decreased through the involvement of other researchers in the analysis of the experiences. We present our experiences from cases in the form of a general method and generally applicable recommendations, which we have constructed from data in the manner of *grounded theory research* [23] and will be detailed in conjunction with the case descriptions. The results should therefore be seen as a valuable generalization of experiences but not yet scientifically validated on additional, independent cases.

The rest of this section presents the cases. Although the systems belong to different domains – automation and power technology domains respectively, having specific focus and facing different issues, the decision was in both cases to transform the existing systems towards product line architectures.

2.1. Case 1

The first case is an industrial automation control system which consists of more than three million lines of C and C++ code. All the source code is compiled into a single binary software package, which has grown in size and complexity as new features and solutions are added to enhance functionality and to support new hardware, such as sensors, I/O boards and production equipment. The software package also consists of various software applications, aiming for specific tasks that enable the automation controller to handle various applications such as painting, welding, gluing, machine tending and palletizing. However, the software package is monolithic, i.e. the complete set of functionalities and services is included in every product even though not everything is required in each specific application. As the system is expanding, it has become more difficult to ensure that the modifications of specific application software do not affect the quality of other applications. The original coarse-grained architecture is depicted in Figure 1.

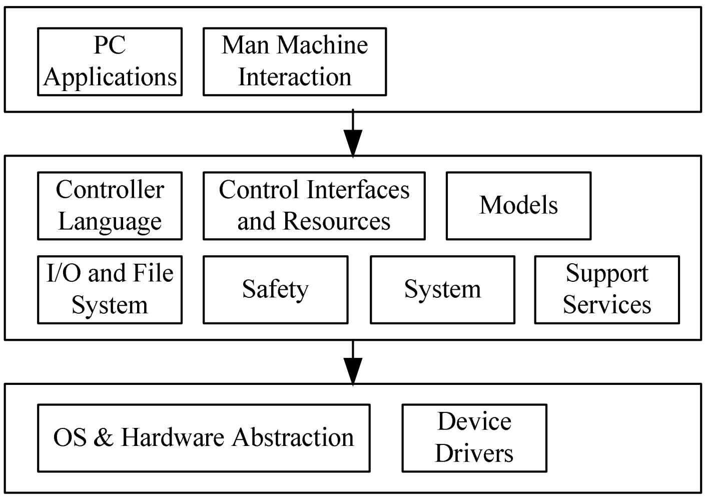


Figure 1. Original Conceptual Architecture

The main problem with the software architecture is the existence of tight coupling between some components that reside in the different layers. As a consequence, source code updates have to be done not only on the application level, but through several layers, several subsystems and components. Recompilation of the whole code base is necessary. This requires that application developers have a thorough knowledge of the complete source code, and additionally, it constitutes a bottleneck in the effort to enable distributed application development. Therefore, there is a need to transform the existing system into reusable components that can form the core of the product-line infrastructure, and separate application-specific extensions from the base software.

2.2. Case 2

The second case is a power control and protection system which consists of more than two million lines of C and C++ code. It is built up from a basic system which handles communication, I/O and services, and from application functions that are combined to define various products. These application functions are built as components for specific functionality in an IEC 1131 fashion, including functions such as monitoring of current and voltages, and control of breakers. The application functions are included in the system builds through definition files, resulting in a specific binary software package for each product. Software development is performed by several different development teams from two separate business units and across different geographical locations. The main problem in this case is not apparently architecture-related as in the first case. It is more related to the product development management problems, i.e. the occurrence of overlapping development functionality, lack of traceability of product features and decreased reusability, as the product variants are implemented in new or version-branched source code files that are scattered in different parts of the code repository. All the projects fetch the base software source code from the repository to start their respective development of various products. The results of the changed software artifacts are not integrated back into the repository. New projects might start and continue from the results from an earlier project and establish new branches of configuration management paths. This leads to additional effort required for maintenance of diverging software and software testing. Therefore, instead of making branches of the core assets for each product variant, there is a need to improve the handling of the common set of core assets through explicit definition of commonalities and variabilities, and build a common platform, from which products can be efficiently developed and launched to the market.

3. Migration Method

The method we devised and used in the two cases is illustrated in Figure 2. It starts with a migration decision, consists of five steps with a proposal for the new architecture and a plan for the implementation/transition process. To explain the steps of the method and demonstrate how the method can be used, we illustrate using the first case as an example; however the method as presented here draws on the experiences from both cases.

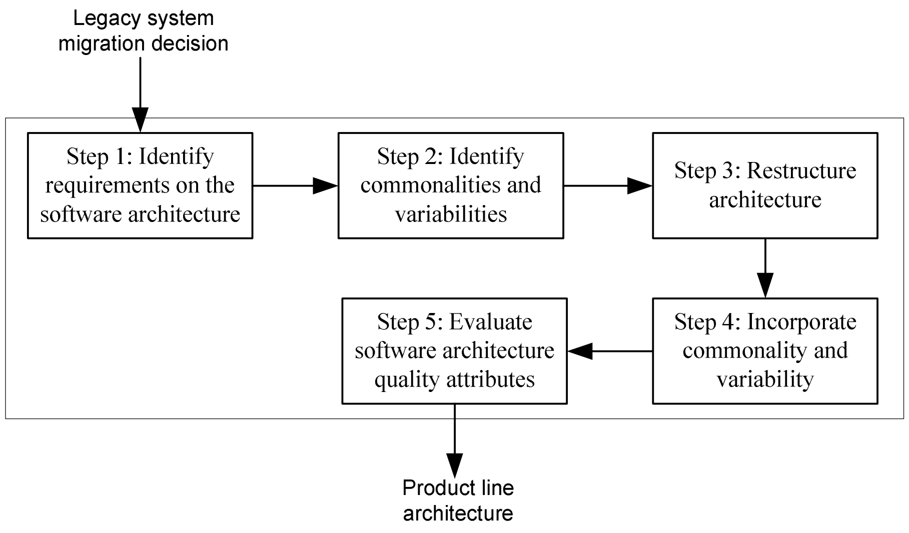

Legacy system migration decision
Step 1: Identify requirements on the software architecture
Step 2: Identify commonalities and variabilities
Step 3: Restructure architecture
Step 4: Incorporate commonality and variability
Step 5: Evaluate software architecture quality attributes
Product line architecture

Figure 2. Migration Method of Legacy Systems to Product Lines

3.1. Step 1: Identify requirements on the software architecture

In this step, requirements essential for a cost-effective software architecture transition to product line architecture are extracted. Architecture workshops need be conducted, where the stakeholders discuss about the underlying business forces for migration, and identify architecture requirements and corresponding migration activities. In order to establish a basis for common understanding of the architecture requirements among the stakeholders within the organization, all the identified requirements need to be prioritized. In the first case, the main focus is to identify components that need to be refactored to facilitate a product line architecture and to define an evolutionary path of the software system development. The identification and analysis of the architectural requirements was performed by the architecture core team consisting of 6-7 persons. We list below the identified main requirements on the software architecture:

R1. More modularized software architecture.

R2. Reduced complexity of the architecture structures.

R3. The architecture needs to support distributed development with minimum dependency between the development sites.

3.2. Step 2: Identify Commonalities and Variabilities

In this step, common core assets and variabilities to facilitate product deployment are identified. The common core asset identification can be based on either a top-down approach, where the product line architecture comprises of union of merged product functionality, or a bottom-up approach where the product line architecture comprises of the functionality shared among the products and exclude product-specific features [4]. There are different ways to identify commonalities and variabilities, e.g. using application-requirements matrix, priority-based analysis and/or checklist-based analysis [18]. The output is a catalog of shared product line assets common for all the applications or products, in terms of requirements, use cases, components and test artifacts.

In the first case, the application-requirements matrix approach was applied, i.e. the dependency analysis between applications and base services was performed to identify commonalities and variabilities. The use of the matrix proved useful as a tool for the architects. Table 1 gives an example of the dependency analysis between specific applications extensions and base services, where x represents the expected presence of a dependency and nothing for its absence.

Table 1. Analysis Matrix Example for Commonalities and Variabilities

	Services					
Application Extensions	alarm	error log	ipc	configuration	device	etc
Arc welding	X	X	X			
Painting	X	X	X	X		
Picking, Packing	X	X	X	X		
etc						

To perform the dependency analysis, sufficient overview of product features is required. The identification of variation points can be based on the architecture description and design documents, source code, compiled code, linked code and running code [Svahnberg et al. 2001], user documentation and user expectations, requirement specifications, log files and comments of changes as well as workshops with concerned development organizations. Accordingly, modules, components and functions that are essential for all applications were identified as candidates for commonalities, designated as included in the *kernel*. Software artifacts that are only mandatory for a small set of applications were identified as candidates for variable artifacts, designated as *common extensions*. The kernel and common extensions form up the building blocks for all applications and they can be packaged into a software development kit (SDK), which provides necessary tools and documentation for application development.

3.3. Step 3: Restructure Architecture

In this step, the product line architecture is constructed. The architecture describes the high level design for the applications of the intended software product line. Architecture workshops need to be conducted, where the architecture core team members and technical leaders in the development projects reach a common understanding of how the entire product line should be structured to fulfill the identified architecture requirements. In the first case, to cope with R3, *the architecture needs to support distributed development with minimum dependency between the development sites*, and the architectural problems described in section 2.1, the strategy of separate concerns was applied to isolate the effect of changes to parts of the system [10]. The strategy was to separate the global functions from the hardware, and separate application-specific functions from generic and basic functions as illustrated in Figure 3.

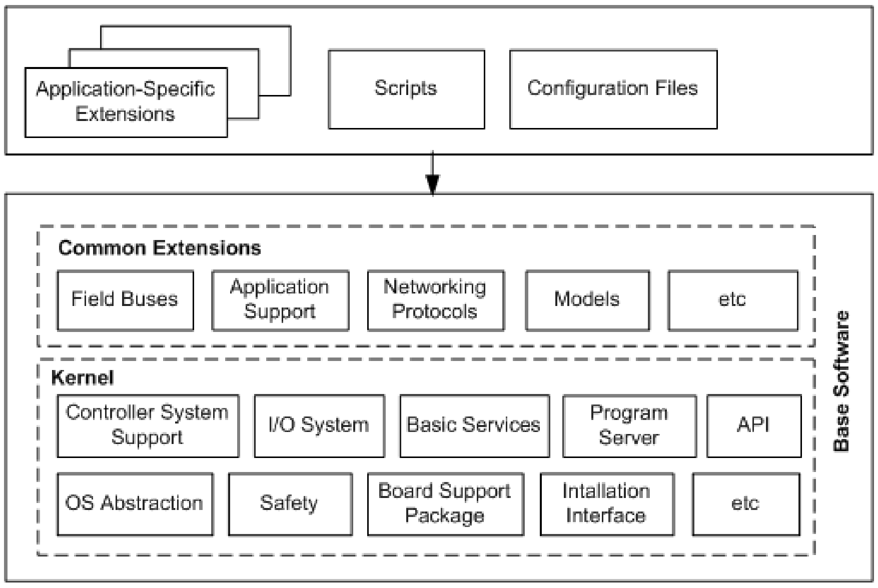


Figure 3. Revised Conceptual Architecture

The identified core assets from the previous step provide input to the definition of global generic functions and application-specific functions. Accordingly, some components need to be adapted and reorganized to enable the restructuring of the architecture. Some examples in the first case were the components for resource allocations within the low-level *Basic Services* subsystem, e.g. semaphore ID management component, and memory allocation management component. These components needed to be adapted because functionality needed to be separated from resource management, to achieve the build- and development-independency between the kernel and extensions.

3.4. Step 4: Incorporate Commonality and Variability

In this step, feasible realization mechanisms and implementation proposals to facilitate the revised product line architecture are defined. Potential refactoring proposals are identified from technical and business perspectives. Technical assessment takes into consideration change propagation and the effect of refactoring, while keeping some important extra-functional properties such as performance or reliability. Business assessment includes the estimation of the cost and effort on implementations. We exemplify with one component example from the first case– the *Inter-Process Communication (IPC)* component that needed to be refactored. IPC belongs to *Basic Services* subsystem and it includes mechanisms that allow communication between processes, such as remote procedure calls, message passing and shared data. We focus on the technical assessment and present the example in terms of three views - problem, concrete requirements and implementation proposal.

Problem: All the slot names and slot identities (ID) used by the kernel and extensions were defined in a C header file in the system. The developers had to edit this file to register their slot name and slot ID, and recompile the system. Afterwards, both the slot name and slot ID had to be specified in the startup command file for thread creation. There was no dynamic allocation of connection slot. The problem was related to requirement R3.

Concrete implementation requirements: It should be possible to define and use IPC slots in common extensions and application extensions without the need to edit the source code of the base software and recompile.

Implementation proposal: The slot ID for extension clients should not be booked in the header file. Extensions should not hook a static slot ID in the startup command file. The command attribute `dynamic slot ID` should be used instead. The IPC connection for extension clients will be established dynamically through the `ipc_connect` function as shown in Figure 4.

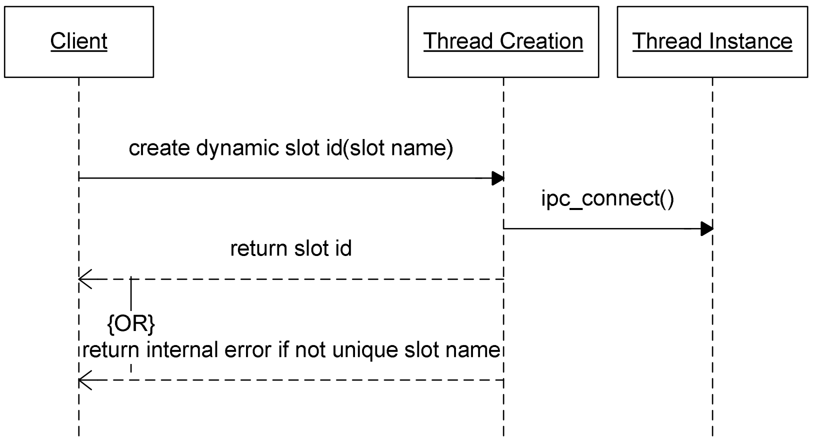

Client
Thread Creation
Thread Instance
create dynamic slot id(slot name)
ipc_connect()
return slot id
{OR}
return internal error if not unique slot name

Figure 4. IPC component after refactoring

3.5. Step 5: Evaluate Software Architecture Quality Attributes

In this step, the impact of implementation proposals on the quality requirements of the product line architecture is evaluated. This is needed as the choice of component refactoring proposals for fulfilling each requirement might lead both to an improvement of some quality attributes, and to a degradation of another quality attribute, which would then require a tradeoff decision. Various assessment techniques [5] can be applied, e.g. scenario-based assessment, software performance assessment and experience-based assessment. Besides the qualitative evaluation, test scenarios and prototypes can also be used as additional ways for evaluating the feasibility and suitability of implementation proposals. In the first case, the experience-based assessment and logic reasoning was applied, and the proposed solutions were evaluated with respect to quality characteristics that were of interest to the stakeholders, i.e. analyzability, changeability, extensibility, testability and real time performance. Table 2 gives an example of the IPC component evaluation.

Table 2. Architectural Consequence Evaluation

	Consequences of changing the Inter-Process Communication
Analyzability	***Degraded*** due to decreased possibility of static analysis because of dynamic definitions
Changeability	***Improved*** due to the dynamism which makes it easier to introduce and deploy new slots
Extensibility	***Improved*** due to encapsulation of IPC facilities and dynamic deployment
Testability	No impact
Real time performance	***Improved*** as resource limitation issue is handled through dynamic IPC connection ***Degraded*** due to introduced dynamism the system performance could be slightly reduced

The revised IPC component provides efficient resource booking for inter-process communication and enables encapsulation of IPC facilities. Accordingly, distributed development of extensions utilizing IPC functionality is facilitated. The use of dynamic IPC connections handles resource limitations, since limited IPC resources are used only when the processes are communicating. However, the use of IPC mechanisms requires resources, which are limited on a real-time operating system. Therefore, the overhead due to resource description processing may be an offset against efficiency [19], since the overall performance may be degraded if the cost of creating and destroying IPC connections is too high.

4. Observations and Recommendations

Applying a software product line approach to legacy systems requires that care is taken to ensure that critical aspects are considered for a smooth and successful product line migration. The application of the migration method provided a structured way to cover these critical aspects and handle the product line transition. Through applying the method in our industrial cases, observations have been made with respect to business, organization, development process and technology when adopting a product line approach. We also use the experiences from the case studies to recommend practices that proved particularly useful.

4.1. Business

We list below observations and recommendations that concern business perspective.

- **Observation: Different triggers for decisions to adopt a product line approach exist**. Business objectives motivate architecture and process changes [15]. The triggers for these changes might appear different although the decision to have product line approach was the same for both case studies. The trigger in the first case was to improve software quality and enable distributed product development. In the second case, the main trigger was to build a common platform that can be shared between two business units and enable component reusability. Our conclusion is that the concept of product lines can be a solution to different types of business goals.

- **Recommendation: Improve risk management through constant progress measuring**. Product line migration concerns a collection of factors [7], such as resources involved, management support and involvement, level of product line expertise, and priority balancing among various projects. A careful and comprehensive risk assessment is therefore necessary. Through the case studies, we observed the benefit of setting up reasonable, achievable, and measurable targets to constantly monitor the progress. For instance, in the first case study, a metric was the number of exposed public interfaces. Constant monitoring of this metric was conducted on a regular interval. It was helpful in measuring progresses and provided signal indication on analyzing the reason for trend of increasing number of interfaces when this happened. This in turn provided a source of input to risk judgments.

4.2. Organization

According to [4], product line development can be organized in two ways: (i) in a separate product line team – one team develops the core assets while other teams develop products; or (ii) within the product team – the development team is responsible for both product and core asset development. Both organization structures were reflected in the two case studies and we observed advantages and disadvantages with both structures. In the first case study, there was one core asset development team centralized at one site and product development teams were geographically distributed. A risk identified for this organizational structure was that the core assets development might not be aligned with the product development schedule. In the second case study, the development of common platform components was part of the concrete product development projects. The development teams were also geographically distributed in several countries. Much focus was on product development, especially when there was a tight schedule on product deliveries. Enhancements and adaptations of platform components were executed in the context of the related product development projects. Accordingly, a risk was reduced reusability of core assets. Another risk was parallel or duplicate development of functions, especially when there are several product development projects running in parallel. However, there is no clear answer on which organization structure is better [6].

- Recommendation: Product managers for different products using the product line architecture should synchronize needs. Our experience in handling the risk in the first type of organization structure was that the product managers need to synchronize to achieve a common understanding of the priorities of product requirements. Synchronization among various product development teams was also required.

- Recommendation: Define roles, responsibilities and ways to share technology assets. The risks for the second type of organization structure was handled through the definition of repository handling strategies, clear ownership of the core assets and clear division of responsibilities for the core asset development. Communication and synchronization between the development teams play a substantial role. For instance, in the second case study, there was a white paper defining the ownership and responsibility areas of existing core assets. Meanwhile, communication channels were open for emerging new functionality and software assets.

4.3. Process

We list below observations and recommendations concerning the process perspective. Additional aspects from case 1 can be found in [15], e.g. regarding configuration management and build processes.

- Recommendation: Perform the migration to product lines through incremental transitions. Despite of the assumption that it requires an upfront investment of 2 to 3 products worth of development effort in order to see return on these investments [7], it is generally required to minimize the upfront investment and to facilitate quick incorporation of product line technology into an organization [26]. In this sense, we assume that incremental transition strategy is a preferred choice to fulfill this requirement without disrupting the ongoing projects. For instance, in the first case study, the criteria for requirement prioritization were set up as: (i) enable building of existing types of extensions after refactoring and architecture restructuring; and (ii) enable new extensions and simplify interfaces that are difficult to understand and may have negative effects on implementing new extensions. Based on these criteria, architectural requirements and components that needed to be refactored could be categorized into different priorities. In addition, one requirement during the component refactoring process in the case studies was to preserve the external behavior of the system despite the number of changes to the code. Accordingly, a sequence of incremental code transformation steps was identified, performed and verified before being integrated.

- Recommendation: Ensure communication between technology core team and implementation team. The vision of migrating legacy systems towards product lines comes quite often from analysis results of a technology core team consisting of very few people. The technology core team needs to communicate the vision on a regular basis with implementation teams, in order to introduce a common understanding and acceptance of what should be accomplished with the transition. The outcome of this is an organization that is informed and prepared for the product line transition process.

4.4. Technology

We list below observations and recommendations that concern technology perspective.

- Recommendation: Use tool support for dependency analysis. Software complexity is due to the inherent complexity in the problem domain and defects in software design [6], e.g. insufficient modularization, which in turn leads to decreased analyzability and changeability. Although the domains of the two cases were very different, the components/modules were not prepared for direct migration in any of the cases. Some components needed to be adapted and reorganized to enable the product line transition. Through the refactoring process, we noticed that coupling and interface definition were two common issues that needed to be handled. We also experienced the need to reduce inter-module dependencies [17], since excessive inter-module dependences in software can make modules hard to develop and maintain. For instance, in the first case, the refactoring solutions were sometimes straightforward and we knew how to refactor with only local impact. When the implementation was uncertain and might affect several subsystems or modules, prototypes were made in order to investigate the feasibility of potential solutions as well as the estimation of implementation workload. In this sense, it would be helpful to have good tool support to facilitate quantitative dependency analysis and impact estimation on workload when making architectural changes.

- Recommendation: Use architecture documentation to improve architectural integrity and consistency. We found out from the two case studies that a strategy for communicating architectural decisions was to appoint members of the core architecture team as technical leaders in the development projects. Although helpful to certain extent, this strategy did not completely prevent developers from insufficient understanding and/or misunderstanding of the initial architectural decisions. This may result in uninformed violation of architectural conformance and lead to architecture quality degradation in the long run. In addition, variation points change during the software life cycle. It is essential to document these changes with respect to what does vary, why it varies and how it varies [Pohl et al. 2005], and to record rationale for each design decision, strategy and architectural solution.

- Recommendation: Carefully define variation points and realization mechanisms. Having pre-determined variation points makes it relatively easy to introduce changes during software evolution [12]. Variation points help to keep the impact of changes small by enforcing separation of concerns among variants. Missing identification of variation points and realization mechanisms in the beginning might lead to extra implementation efforts later. For instance, in the second case, operation data could be transferred over a number of communication protocols, such as IEC 61850, IEC 60870, LON, DNP, and Modbus. However, the mechanism to facilitate this variability was missing. This resulted in extra efforts for adding new communication protocols and additional amount of rework for modifying existing ones.

On the other hand, we need to consider the impact with respect to the software system's behavior, quality and any possible tradeoffs when we introduce any variation point and realization mechanism. For instance, the choice of binding mechanisms and binding time has consequences for flexibility and other concerns [8]. In the second case, the original architecture applied 'reduce computational overhead' principle, which resulted in inclusion of several application functional components in the base software and making direct calls to them instead of using an intermediary layer. The reason for this was mainly performance related. This became a performance versus modifiability tradeoff point.

- Recommendation: Use the described method iteratively to handle software evolution. Software evolves as well as businesses and environments. It is therefore necessary to iterate over the five steps during the software lifecycle when certain decisions need to be made, e.g. to determine if any new features added to a product should be incorporated into the product line architecture or restricted to the particular product.

5. Related work

Software product line has emerged as one of the dominating paradigms for cost-effectively developing software products. A great amount of research has been done in this area. Bosch [Bosch 2000] proposes methods for designing software architecture, in particular product line architecture. Pohl et al. [Pohl et al. 2005] elaborated two key principles behind software product-line engineering: (i) separation of software development in domain and application engineering, and (ii) explicit definition and management of variability of the product line across all development artifacts. A four-dimensional software product family engineering evaluation model is described in [27] to determine the status of software family engineering concerning business, architecture, organization and process. Our observations are classified into similar dimensions.

Faust et al [9] presented metrics for genericity relayering, and migrated multiple instances of a single information system to a product line. The idea of constructing a federated architecture was similar to the way that we have performed in our case studies.

Bayer et al [1] presents the RE_MODEL method to integrate reengineering and product line activities to achieve a transition into a product line architecture. A key element in the method is the *blackboard*, a work space which is shared for both activities that are done in parallel. This is similar to the way that we have performed in our case studies, with a common repository for all information, both for reengineering activities and for product line activities.

A case where a component was refactored to fit into a product line context was presented by Kolb et al in [Kolb et al. 2005]. The PuLSE^{TM} method was used to systematically analyze the component and to improve its reusability as well as maintainability. The focus was on one component enabling reuse of that component. The usage of PuLSE in an embedded environment was described in [21], where the method's technical components addressed the different phases of product line development. Our approach focuses on the migration process when the migration decision has been made. In [25], the FODA method [11] was used for domain engineering whereas we applied product modeling in our method. In order to evaluate the potential of creating a product line from existing products, MAP (Mining Architectures for Product Lines) was described in [22], which focuses on the feasibility evaluation process of the organization's decision to move towards a product line. Options Analysis for Reengineering [3] is another method for mining existing components for a product line. [16] describes combining reference architecture and configuration architecture to describe legacy product family architecture and manage its evolution.

6. Conclusions and Future Work

In this paper, we presented our product line migration method which was devised through our participation in two industrial migration projects. Throughout the use of the method, the architecture requirements and corresponding design decisions for the transition towards product line architecture become more explicit, better founded and documented. The resulting documentation of refactoring proposals was in the cases widely accepted by the stakeholders involved in the migration process. Our experiences shows the importance of synchronizing needs, defining roles, communication between core team and implementation team for architectural integrity, and using proper tools for dependency analysis. Also, the business and process contexts require the transition to be incremental, and the architecture therefore needs to support this through explicit definition of implementation proposals.

Our plans are to apply the migration method in new cases and in new domains, and collect additional experiences in product line migration.

This work was partially supported by the Swedish Foundation for Strategic Research (SSF) via the strategic research centre PROGRESS and by the KK-foundation (KKS) through the SAVE-IT project.

References

[1] Bayer, J. Girard, J. F., Wûrthner, M., DeBaud, J. M. and Apel, M., "Transitioning legacy assets to a product line architecture," Proceedings of the 7th European Software Engineering Conference. Toulouse, France, Springer, 1999.

[2] Bennett, K. and Rajlich, V., “Software Maintenance and Evolution: a Roadmap”, 2000.

[3] Bergey, J. O'Brien, L. and Smith, D., "Using options analysis for reengineering (OAR) for mining components for a product line", Proceedings of Second Software Product Line Conference, volume 2379, pp. 316-327. Springer, 2002.

[4] Birk, A. Heller, G. John, I. and Schmid, K. et al, "Product Line Engineering: The State of the Practice," IEEE Software, 2003.

[5] Bosch, J., *Design and use of software architectures: adopting and evolving a product-line approach*, ACM Press/Addison-Wesley Publishing Co., 2000.

[6] Bosch, J., "Product-Line Architectures in Industry: A Case Study", ICSE 1999.

[7] Clements, P. and Northrop, L., *Software Product Lines: Practices and Patterns*, Addison-Wesley Professional, 2001.

[8] Coplien, J., *Multi-Paradigm Design for C++*, Addison-Wesley, Boston, Massachusetts, 1998.

[9] Faust, D. and Verhoef, C., "Software product line migration and deployment", Journal of Software Practice and Experiences, 33(10):933955, Aug. 2003.

[10] Hofmeister, C., Nord, R. and Soni, D., *Applied Software Architecture*, Addison-Wesley, 2000.

[11] Kang, K. C. et al, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", Technical Report CMU/SEI-90-TR-21, SEI, Carnegie Mellon University, 1990.

[12] Kolb, R., Muthig, D., Patzke, T. and Yamauchi, K., "A Case Study in Refactoring a Legacy Component for Reuse in a Product Line," Proceedings of ICSM '05, 2005.

[13] Kolb, R., Muthig, D., Patzke, T. and Yamauchi, K., "Refactoring a legacy component for reuse in a software product line: a case study," Journal of Software Maintenance and Evolution: Research and Practice, vol. 18, pp. 109-132, 2006.

[14] Kotonya, G. and Hutchinson, J., "A Component-based Process for Modelling and Evolving Legacy Systems", Software Process: Improvement and Practice, 13(2), pp. 113-125, 2008.

[15] Larsson, S. Wall, A. and Wallin, P., "Assessing the Influence on Processes when Evolving the Software Architecture," Proceedings of IWPSE 2007, Dubrovnik, Croatia, 2007.

[16] Maccari, A. and Riva, C., "Architectural evolution of legacy product families", Proceedings of the Fourth International Workshop on Product Family Engineering, 2001.

[17] Parnas, D. L., “Designing Software for Ease of Extension and Contraction”, Transaction on Software Engineering, SE-5(2), 1979.

[18] Pohl, K. Böckle, G. and v. d. Linden, F. J., *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, 2005.

[19] Quecke, G., Ziegler, W., "Mesch - an approach to resource management in a distributed environment", Proceedings of the First IEEE/ACM International Workshop on Grid Computing. Springer-Verlag, pp. 47–54. 2000.

[20] Ransom, J., Sommerville, I. and Warren, I., "A Method for Assessing Legacy Systems for Evolution," presented at Reengineering Forum '98, Florence, Italy, 1998.

[21] Schmid, K., John, I., Kolb, R. and Meier, G., "Introducing the PuLSE Approach to an Embedded System Population at Testo AG", ICSE, 2005.

[22] Stoermer, C. and O'Brien, L., "MAP - mining architectures for product line evaluations", Proceedings of WICSA'01, pages 35-44. IEEE Computer Society Press, Aug. 2001.

[23] Strauss, A. and Corbin, J. M., *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory* (2nd edition), ISBN 0803959400, Sage Publications, 1998.

[24] Svahnberg, M., Gurp, J. V. and Bosch, J., "On the Notion of Variability in Software Product Lines," Proceedings of WICSA'01), Amsterdam, The Netherlands, 2001.

[25] Thiel, S., Ferber, S. et al., "A Case Study in Applying a Product Line Approach for Cae Periphery Supervision Systems", Proceedings of In-Vehicle Software, SP-1587, pp. 43-55, 2001.

[26] Tilley, S., "The Net Effects of Product Lines," in SEI Interactive, 1999.

[27] van der Linden, F., Bosch, J., Kamsties, E., Kansala, K. and Obbink, H., "Software Product Family Evaluation", Proceedings of SPLC, 2004.