Mälardalen University Press Licentiate Theses
No. 97

SOFTWARE ARCHITECTURE EVOLUTION AND SOFTWARE EVOLVABILITY

Hongyu Pei Breivold

2009

School of Innovation, Design and Engineering

Copyright © Hongyu Pei Breivold, 2009
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 .... **1**

Chapter 1. Introduction **3**

1.1 Research Motivation 4
1.2 Research Questions 5
1.3 Thesis Overview 8

Chapter 2. Research Results **13**

Chapter 3. Research Method **21**

3.1 Research Process and Method 22
3.2 Validity Discussions 24

Chapter 4. Related Work **31**

4.1 Software Evolution 31
4.2 Software Quality Models 34
4.3 Software Process Models 38
4.4 Software Architecture Evolution 42
4.5 Software Architecture Evaluation 44
4.6 Component-Based and Service-Oriented Software Engineering 47
4.7 Software Product Line Engineering 48
4.8 Reverse Engineering and Reengineering 52
4.9 Software Quality Metrics 54

Chapter 5. Conclusions and Future Work **57**

5.1 Contributions 57
5.2 Future Research Directions 58

References **61**

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.

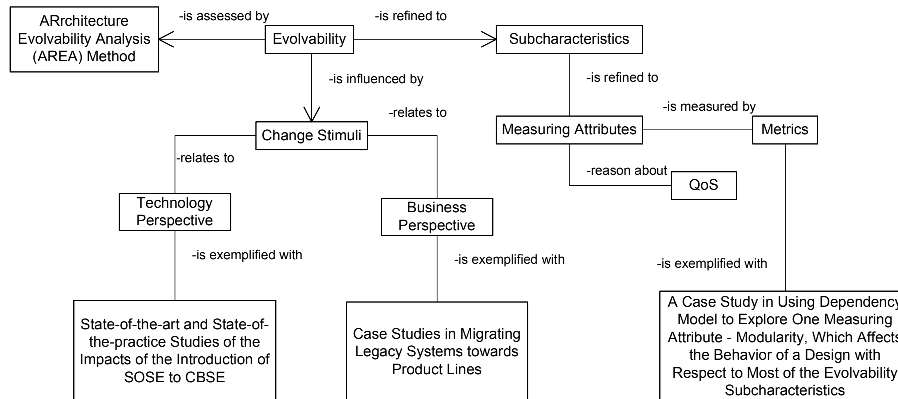

Figure 1. Contributions of the Thesis

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™ 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